Fachbereich Informatik

Symbolic Timing Diagrams: A Visual Formalism for Model Verification

Rainer C. Schlör

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften
des Fachbereichs Informatik
der Carl-von-Ossietzky Universität
Oldenburg

Gutachter:	Prof.Dr. Werner Damm Prof.Dr. Wolfgang Thomas
Tag der Disputation:	3. März 2001

email: Rainer.Schloer@ewetel.net

To my family

Contents

Acknowledgements

The work described in this thesis builds alongside the results and contributions of many people. First of all, I would like to thank Werner Damm for providing the basic ideas about using timing diagram in a formal way which finally lead to the developments presented here. He also shaped my first publications on this subject. I am also grateful to Wolfgang Thomas for many valuable comments on this work, in particular on the logic part of the thesis.

The work would not have been possible without the prior development of a framework for compositional reasoning based on the temporal logic MCTL by Bernhard Josko, who also took part in several extensions of this work and was helpful at any time when I needed sophisticated expert advice.

While the years passed by, I worked in different teams. Most enjoyable was the cooperation during the FORMAT project with several people [1]: Massimo Bombana, Patricia Cavalloro, Alberto Allara (Italtel, Milano), Bob Harris, Simon Finn, Chris Read, Colin Saunders (Abstract Hardware, London), Thomas Filkorn, Ronald Herrmann, and Jörg Bormann (Siemens ZT, Munich). I also enjoyed a follow-up project with Dieter Werth (Siemens AT, Nürnberg).

Around the same time, Franz Korf developed a system for interface controller synthesis (ICOS) from a dialect of temporal logic, which was inspired by the semantics of STD presented in this work. His work was an important building block for the vision how STD could be practically used.

Another important cooperation was performed with Johannes Helbig, who developed an axiomatic semantics for Statecharts using STD-idioms. One goal of his work was to provide a basis for compositional verification of Statecharts with STD. This work came to a practical implementation during the past years with the development of a verification framework for Statemate™.

Most of the implementation work which was needed to embed STD into a practical framework for property specification (including graphical editor design, compiler and database integration) was performed by Hartmut Wittke, with important building blocks (in particular abstract datatypes) contributed by Ingo Schinz. Without their work, STD would probably not have survived until the time of writing of this book.

Special thanks are due to Henning Dierks for his careful reading of this thesis, which allowed me to fix many typos and subtle problems before publication. Also thanks to Martin Fränzle for a great lecture on duration calculus and related issues.

Finally , I thank my family for their patience, while waiting for this work to be completed. Like many other researchers, I had to spend many weekends and some nights out of my home. I would never have managed without their support, in particular from my wife.

Oldenburg, January 24, 2002

This thesis was typeset with LaTeX by the author.

[1] This work has partially been funded by ESPRIT project No.6128 (FORMAT).

Zusammenfassung

In der vorliegenden Arbeit wird eine neuartige visuelle Spezifikationtechnik mit der Bezeichnung STD (Symbolic Timing Diagrams) eingeführt und untersucht. Die Spezifikationstechnik wurde speziell für die Verifikation von Modellen reaktiver Systeme entwickelt.

Die Arbeit hat drei Teile: Zunächst wird eine Einführung und Motivation zur Verwendung von STD gegeben. Im zweiten Teil der Arbeit werden die theoretischen Werkzeuge und Begriffe eingeführt, auf deren Basis die Semantik von STD beruht. Der Formalismus STD wird in zwei Stufen eingeführt und erklärt: Zunächst wird eine syntaktische Untermenge von STD (mit der Bezeichnung LSTD) analysiert, und das Verständnis der Semantik von LSTD durch die Vorstellung eines Satzes von Beweisregeln (Ableitungsregeln) unterstützt. Im zweiten Schritt wird der Hauptformalismus STD definiert. Der Zusammenhang zwischen LSTD und STD wird durch einen Hauptsatz hergestellt, der zeigt, daß sich jede STD Spezifikation in eine äquivalente LSTD Spezifikation überführen läßt.

Obwohl die Arbeit sich auf die theoretischen Grundlagen von STD konzentriert, wurden zahlreiche ergänzende praktische Studien seit 1995 durchgeführt und das Konzept von STD bis zur Entwicklung eines in der Praxis anwendbaren Werkzeugsatzes fortgeführt. Dies wird kurz im ersten Teil der Arbeit beschrieben. Der Formalismus wurde in verschiedene Verifikationsumgebungen integriert, insbesondere für die bekannten Modellierungssprachen VHDL und Statemate™.

Summary

The work described in this thesis introduces a novel visual specification formalism named STD which can be used to verify models of reactive systems.

The presentation has three parts: First, an introduction and motivation is given. Second, a theoretical framework needed to describe the context of the design of STD is layed out. Third, the novel formalism itself is introduced in two steps: First, a syntactical subset (LSTD) of STD is analyzed in detail, and the semantics is illustrated by a representative set of derivation proof rules. Second, the main formalism STD is defined. The connection between LSTD and STD is established by a theorem, which shows that a STD specification can be represented by an equivalent LSTD specification.

While the presentation is focused on the theoretical foundation of STD, complementary work has been performed over the past five years to bring the concept of STD to a usable implementation. This is shortly described in the first part of the thesis. The formalism has been integrated into existing verification frameworks for different specification languages, in particular for VHDL and the Statemate^{TM} framework.

Chapter 1

Introduction

The introduction outlines the motivation and the decisions taken for the work described in this book.

First, the goal of the work is characterized. The following sections discuss some fundamental issues related to the topic of verification and logic, as well as a discussion of related work as far as it is relevant for this thesis.

1.1 The goal

The title of this book is: "Symbolic Timing Diagrams: A visual formalism for model verification". From the title, the following keywords need an explanation:

- Symbolic
- Timing Diagram
- Visual formalism
- Model, and
- Verification.

We start with the explanation of the familiar keywords. The concept and notion of *Timing Diagrams* has been used for a long time. It refers to a diagrammatic representation of the values on a set of wires observed over time. In hardware, the value of a wire is an analogue voltage level. In logic, the value is one of a set of logic values, e.g. high ('1'), low ('0'), unknown ('X'), or high–impedance ('Z').

Usually a (strictly) monotonic increasing progress of time is assumed along the X–axis of a timing diagram. The according values on a wire over time are depicted in so–called *waveforms*. Several wires with their associated waveforms are depicted in a timing diagram. The wire–names and the waveforms are allocated along the Y–axis of a timing diagram.

The advantage of such a visual representation is given by the fact that a complex interplay of values over time on a set of wires can be understood easily from the diagram. A corresponding representation in textual (e.g., tabular) form would be much harder to read. A typical application for timing diagrams is the presentation of bus protocols or parts thereof (e.g. read–cycle, write–cycle etc).

This brings us to the next keyword. *Visual formalism* is a term referring to a special type of formalism, which has a mathematical rigorous notation. The standard definition of a formalism is by means of a grammar. The derivations of a grammar lead e.g. to formulas. Formalisms are simpler than programming languages, which involve further concepts and notions such as variable and function declarations, block structure, and scope.

A *visual* formalism can be defined by a grammar, which shows how the graphical objects are derived. It could also be defined by any other type of definition; the point is that from a given diagram the constituents can be unambiguously distinguished and that a formal semantics can be assigned to the constituents.

The word *Model* is a term with a broad meaning. In this book we use the term in the context of a development process, which starts from high–level requirements, and ends at a "golden device", i.e. a description of a system, which can be translated (mostly) automatically into code. E.g., for a subset of the hardware description language *VHDL* [2], synthesis can be performed. Similarly, code generators exist for designs developed in the *Statemate* environment [20].

In reality, a model is almost never complete or even accurate; in the best case, it is useful. This thesis relies on the assumption that models are useful during the development process, and worthwhile to be constructed with the highest possible precision.

The consideration of the term *verification* will be postponed to the following section.

The probably most unfamiliar term in the title of this book is *Symbolic*. Like the term *model*, has the term *symbolic* a broad meaning. A symbol — or, more specifically, a *(symbolic) expression* – is a compact denotation of another object or set of objects. For instance, the expression $x < 5$ (where x denotes a natural number) can be used as a compact notation for the set $0, 1, 2, 3, 4$. The

(infinite) set of points constituting a circle of radius R around the origin of the 2–dimensional plane of real numbers can be described as the set of points (x, y), which are a solution to the formula $x^2 + y^2 = R^2$.

Here, the attribute "symbolic" is used as part of the more specific notion of a *symbolic waveform*. Recall that in traditional timing diagrams waveforms show the values on a wire over time, which is for instance a sequence of high–, low– and don't–care values. A symbolic waveform generalizes this concept as follows: Each region of the waveform is associated with a Boolean expression. For instance, the Boolean constant *true* corresponds to a don't–care region. The special case of a value sequence on some signal *sig* is represented by a sequence of assertions (Boolean expressions). For example, a value sequence for a Bit–valued signal *Request* could be described by a sequence of assertions as follows: *Request* = '0', *Request* = '1', *Request* = '0', etc. Graphical examples of this idea are given in chapter 2.

Why this book was written. For any research endeavour, there is always at least one incentive:

- An open (unsolved) question
- An open issue (e.g. a manufacturing problem)
- Some other incentive (e.g. need of a handbook on a particular topic)

In the case of this thesis, the starting point was an issue: How can we pave the way to formal model verification? Given the fact that testing is the common industrial practice of verification, formal (or mathematical) verification is a new issue in industry, which just starts to gain attention. In particular, the invention and facilitation of a technique known as model–checking [8] has been a major breakthrough for the idea of formal verification of industrial–scale applications.

However, model–checking relies on a description of (safety–critical) requirements stated in a logical formalism. This turns out to be unacceptable given the current state of experience of a "normal" designer.

Thus, the challenge behind this thesis can be stated in one sentence as follows: Given the fact that (traditional) timing diagrams have been in widespread use for a long time, build a visual formalism that resembles timing diagrams, and at the same time can be used as an input language for model–checking. This thesis introduces an approach to fulfil this wish and explores the consequences following from the invented definitions. It builds on a vision expressed in [11], stated over 10 years ago.

No open questions are solved in this thesis, except for the questions which originate from the definitions themselves. Thus the material is of limited interest for the scientific community. On the other hand, the idea explored in this thesis has — thanks to an early implementation acquired during an ESPRIT project in collaboration with several industrial partners, in particular with SIEMENS R&D — attracted an astonishing amount of attention by industrial partners, who were interested in trial uses (in preparation of a possible development process improvement) of a verification framework. This trend has recently increased thanks to the availability of a verification module (see [6]) available for the Statemate design environment.

1.2 Verification versus Validation

In a nutshell, the term *Verification* means: Building the system *right*, i.e. flawless with respect to written statements (requirements). Thus, verification is a technique to reassure that particular work is done correctly. For instance, a theorem prover can be used to verify a complicated mathematical proof. In fact, (subtle) errors have been found in published (and thoroughly reviewed) mathematical proofs.

Given the existence of a golden device in a development process, i.e. of a model which incorporates all requirements and fills in as many details as needed to obtain running code, verification becomes an important issue. It is used to ensure that all requirements are captured correctly in the model.

Model–checking can be used as a mostly automatic method of verification. Two basic prerequisites are needed in order to apply model–checking:

1. The requirements need to be cast into a formalism (which should be done as "smooth" as possible)

2. The language used to describe the model — e.g. Statecharts — needs to be compiled into a finite–state–machine (FSM).

The main practical problem with verification is model–complexity. Even with sophisticated encoding techniques, the maximum size amenable to verification using model–checking is the size of (major) building blocks of an ASIC or of an embedded system in automotive controller (e.g. a central locking system in a car). Verification can be applied also to full designs using either abstraction techniques or techniques of compositional reasoning. The latter topic will be further expanded in the second and the fourth chapter.

By contrast, the term *Validation* means: Building the *right system*, i.e. using the correct total conception. Validation touches on the question, whether

the requirements are correctly stated and consistent, whether assumptions (regarding system integration) are valid and so on.

Sometimes the terms verification and validation are confused, although validation is in fact something quite different from verification. It can only take place if the complete real system is build, consisting e.g. of a controller, sensors/actuators and the physical environment.

Thus the role of verification is well defined with respect to its duties and limits. Verification greatly emphasizes a modular conception of system architecture, and — with respect to model–checking — relies on clean interfaces and well–structured design style.

1.3 "Visual" versus sentential logic

Mathematicians and engineers work with visual representations quite often. Mostly, diagrams and pictures are used to illustrate a thought, or to efficiently depict a situation, which is awkward to explain in words.

The reverse idea — *using a visual representation in the first place* for an explanation or even specification — is rather seldom used, probably because of the inherent risk of ambiguity in pictures.

The work described herein is based on the assumption that a direct visual representation is useful for requirement specification. Even experts in the field of formal specification (e.g. L.Lamport as well as Z.Manna and A.Pnueli) have experimented with this idea.

The lessons learned by the author of this book over several years are not conclusive regarding the question whether visual formalisms offer a real advantage in the context of formal verification. On the one hand, they appeal to an immediate understanding, which is why even designers without any background in formal methods jump on the idea of formal verification using a graphical input format for specification.

On the other hand, *requirement specification is an inherently difficult task*. There is no such thing as an easy way to requirement specification. All one can hope for, is to get a designer started on the idea of applying formal verification in the everyday work. Once results are seen (e.g. some nasty error found, which was overlooked even after intensive model review), the commitment to further training on formal methods is increased.

In summary, the freedom of choice between "over–visualisation" and "over–formalization" appears to be smaller than expected. If a visual formalism is applied with care, then there is definitely a benefit both for documentation and for specification maintenance.

1.4 Related work

Closest to the work described here is the book [22] by Cheryl Kleuker on *Constraint Diagrams* (a visual formalism developed by herself). Her book also contains a comprehensive account on related work on timing diagrams and other visual formalisms as known until 1999.

At the time where the fundamental decisions regarding the basic definitions of STD were taken (which were published in [32] and in the final form in [13]), there existed only isolated publications about using timing diagrams for formal verification, in particular using model–checking. Closest were the publications [16] and [3], but none of the publications known by that time took into account a concept for compositional reasoning.

Therefore, the structure and the semantics definitions of STD are really novel inventions. The only real influence came from the famous "UNITY" book by M.Chandy and J.Misra [9], which had an impact on many researchers during the 90's. Readers familiar with the UNITY–logic will find correspondences in the design of the sub–formalism LSTD presented in chapter 5.

1.5 Structure of this book

This book is organised into seven chapters. The rest of this introduction provides a short characterization for each of the following chapters.

An introductory chapter on Symbolic Timing Diagrams (chapter 2). Symbolic Timing Diagrams (STD for short) are the main objects of investigation of this thesis. Their syntax and intuitive semantics is illustrated by an example, which shows how requirements are specified in STD. This chapter also gives a sketch of two existing verification environments, which can be used together with STD to verify models. The first verification environment is for designs written in VHDL or Verilog, the second is an extension of the Statemate^{TM} design environment.

An introduction to symbolic automata and temporal logic (chapter 3). The semantics of STD can be expressed in terms of (discrete time) temporal logic. This is the key to applying model–checking of finite–state–machines obtained from designs written in some high–level design language such as VHDL.

The semantics of STD is not defined directly in terms of temporal logic, but by a two–step definition: First, an (automaton based) intermediate format

called *symbolic automata* is used to define the semantics of the body of a (symbolic) timing diagram. Second, the semantics of diagram–activation is defined directly in terms of temporal logic.

This chapter shows that for a subset of SA (partially ordered SA), which suffices to characterize the semantics of diagram bodies, the semantics can be characterized by temporal logic formulas.

Theoretical foundation of model construction (chapter 4). This chapter introduces a model which can be used as foundation of the methodology of compositional reasoning. The exposition follows essentially the approach of the book [26], with only little customization.

In this chapter we also introduce a running example for the rest of the book, which is used for illustration both of specification and of verification issues.

LSTD: Linear (Symbolic Timing) diagrams and their semantics (chapter 5). The traditional reading of timing diagrams assumes a homogenous progress of time along the X–axis of a timing diagram. This is in contrast to the semantics of STD, where the waveforms rather correspond to concurrent processes of a Petri–net: The progress of time is monotonic on each waveform, but can occur at different speed along each waveform of a STD diagram.

Linear (Symbolic Timing) diagrams consist of exactly one waveform. In this case, the classical linear interpretation of the progress of time is preserved.

This chapter investigates the semantics of LSTD in detail, and presents a set of derivation rules which could be used as foundation for an embedding of LSTD into an interactive verification environment (theorem prover).

Definition of STD, and the relation to LSTD (chapter 6). In this chapter, a formal definition of STD diagrams and their semantics is given. Although the definition of linear STD diagrams seems like a very strong restriction compared to STD, it is shown in this chapter that each STD diagram can be "decomposed" into a set (conjunction) of linear diagrams with equivalent semantics. Thus, the general definition of STD does not add expressiveness to the concept of linear diagrams.

Resume and direction for further research work (chapter 7). The book is concluded by a summary with pointers to related research work and suggestions for further extension of the results obtain in this thesis.

Chapter 2

Methodology of model verification

The second chapter aims to give the reader an overview picture of the approach of Symbolic Timing Diagrams (STD). Several items are touched: Methodology of model verification based on model–checking, examples of informal requirements (taken from the PCI–bus specification) and their formalization, the semantics of STD–diagrams, and two examples of verification environments, which have been implemented on top of standard design environments for VHDL respectively Statemate, integrating STD as method for property specification.

2.1 Methodology

Formal verification should be part of a development process, which guides the construction of (safety–critical) software– or hardware–systems through a sequence of steps. Various models have been proposed for development processes. The most familiar model is the waterfall–model, which consists of the following sequence of phases [29]:

- Requirements Analysis
- *Requirements Definition*
- *Design*
- Coding
- Testing, and

- Maintenance.

In software development, the results of all development phases are validated by reviews and inspections. Clearly, reviews and inspections are not infallible; subtle errors are likely to be overlooked. Therefore it is helpful to use automatic verification tools to ensure that the result of a development phase is correct, i.e. conforms with the requirements and results of the previous phase.

The method suggested in this book is applicable to verify that a (prototype–)model developed in the design phase conforms to the requirements stated in the requirements definition phase. It assumes that the semantics of the model developed in the design phase can be expressed by a finite–state machine (FSM), e.g. by a classical Moore– or Mealy–automaton.

The emphasis of this work is on the verification of models of *reactive systems*, i.e. of systems whose behaviour is characterized by continous interaction with their (physical or logical) environment. Examples of reactive systems are embedded controllers, operating systems, and specifically bus–controllers.

Controllers of real–life embedded systems — for example, an electronic niveau–controlling sub–system in a car — grow large and complex. Sometimes this happens even in the early phases of model development, where not all details of the final design have been considered yet.

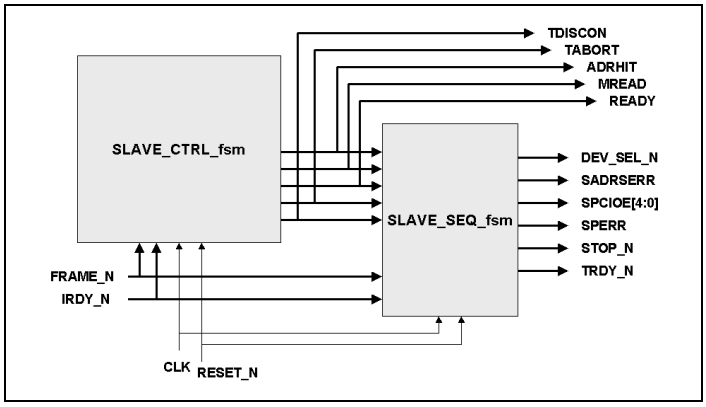
Controllers are usually represented by a parallel composition of interacting state–machines. While the representation of such a model using parallel composition is compact and well suited for simulation, it must be elaborated in order to perform verification. Elaboration means that each possible interacting behaviour must be represented.

The complexity of such a representation grows exponentially in the number of the parallel system components. An attempt to verify a system as a whole suffers from the well–known *state–explosion problem*, which means that a representation of the full computation tree is not possible. This happens with most real–life full designs.

There are two approaches to address the problem of state–explosion. In this book, we favour a methodology which is based on a divide–and–conquer strategy: Verification of a large design is done by specification of the interfaces of its sub–components, and requirements stated for the full design (represented by a parallel composition of its sub–components) are derived logically from the specification of the sub–components.

There is also another approach, which has proved to be very efficient. Recent work has demonstrated that *abstraction* on the model of a full design also allows to verify properties of a complex design. Essentially, both approaches are related and can be used with the model–checking approach and

Figure 2.1: Interface signals of PCI–target sequencer and backend.



the specification formalism STD in the same fashion. Application of abstraction techniques to verify large designs using STD for requirement specification is discussed in [5].

2.1.1 Example: PCI–interface

As running example for this chapter we use a verification study, which has been performed in co–operation with Siemens–AT. The results of this work are summarized in [33].

The focus of this verification study was on the PCI–interface of a bus–bridge (the PMIO–bridge). The specification of the PCI–interface using STD has also be considered in the context of automatic controller synthesis (see [23]).

The PMIO–ASIC consists of several units, each of which implement specific functionalities; examples are: an arbitration unit, a timer unit, and an interface to a PCI–Bus. The PCI–interface implements part of the standardized functionality of a PCI–target unit, defined in [28] (the implementation described here uses the term *slave* instead of the term *target*).

A PCI–target consists of a sequencer unit and a so–called backend. Correspondingly, there are two units called *SLAVE_SEQ_fsm* and *SLAVE_CTRL_fsm* in the PMIO–design, which implement the respective functionality. The relevant interface signals of these two units are shown in figure 2.1.

Both components are synchronous, triggered by signal CLK. We consider

the following informal requirement (more requirements will be stated in the next section):

After STOP_N *has been asserted,*

then it can be de–asserted only if FRAME_N *is de–asserted.*

Without going into the detailed meaning of this statement, it is clear that a relation on the behaviour between signals STOP_N and FRAME_N is stated. Figure 2.1 suggests that it may suffice to consider the behaviour of the sequencer unit *SLAVE_SEQ_fsm in isolation*. This means that the inputs signals TDISCON, TABORT, ADRHIT, MREAD and READY , which are in fact driven by the unit *SLAVE_CTRL_fsm* , are virtually cut and driven by whatever faulty test sequences.

In the ideal case, a system–component (unit) is designed *in a robust way* with respect to critical requirements. Even faulty input stimuli should not be able to affect the required properties of the input/output behaviour of the unit. In real–life, however, this ideal case is seldom found. Much more common is the situation, that certain assumptions (sometimes undocumented) are made in the design of a unit with respect to the values observed on input signals (which in turn may depend on the values observed on output signals of the unit). For example, the VHDL–design considered in this section turned out to be designed in a less robust way than expected. In particular, the unit *SLAVE_CTRL_fsm* appeared to have no independently meaningful behaviour.

The next step was to consider the parallel composition of the two units *SLAVE_SEQ_fsm* and *SLAVE_CTRL_fsm*, which were apparently designed to work together. These two components incorporate still a fraction of the total complexity of the ASIC, and can be verified using the automatic verification method described later in this chapter.

These observations suggests a methodology that can be summarized as follows:

1. Given a property, determine the set of signals referred to in the property.

2. Determine the minimal subset of units, whose composition may be able to guarantee the required property, and perform verification.

3. If the subset of considered units does not suffice, add further components to the composition, as far as complexity issues permit, and perform verification again. Repeat step 3, if necessary.

This methodology can be automated based on data–flow–analysis. Another

Table 2.1: Requirements T1–T5. The numbers in (...) brackets refer to the corresponding operating rule numbers used in [28].

Name	*Description*
T1	Signal DEVSEL# must be asserted at latest at the point, where the target asserts the signals STOP# and TRDY# for the first time (17).
T2	After DEVSEL# has been asserted, then it can be de–asserted only after the end of the last data phase – except in order to signal target–abort (18).
T3	After STOP# has been asserted, then it can be de–asserted only if FRAME# is de–asserted (12b).
T4	If FRAME# has been de–asserted, then STOP# must be de–asserted thereafter (12b).
T5	After one of the signals TRDY# and STOP# has been asserted, then the signals DEVSEL#, TRDY# and STOP# may not be changed before the end of the current data phase (12c).

methodology (based on a divide–and–conquer approach) will be described in detail in chapter 4.

2.2 Requirements

The requirements of a bus interface are particular in the sense that they can be stated "anchored" at well–defined start–conditions (e.g. the bus being in a logical *idle*–state). Moreover, the start–conditions can be explicitly represented as logical expressions over the set of interface signals.

In the following (table 2.1 and 2.2), a number of informal requirements (taken from [28]) referring to the interface signals explained above are stated and named.

In the next section, some of these requirements will be formalized using the visual formalism of STD–diagrams.

Table 2.2: Requirements T6–T10.

Name	*Description*
T6	After STOP# has been asserted, then TRDY# must be de–asserted, after data has been transferred.
T7	After STOP# has been asserted, then TRDY# and DEVSEL# may not be asserted until the end of the last transaction phase.
T8	After the last transaction phase, DEVSEL#, TRDY# and STOP# must be de–asserted.
T9	If DEVSEL# is de–asserted in order to indicate target–abort, then signal STOP# must be asserted and TRDY# must be (or remain) de–asserted.
T10	The signals DEVSEL#, TRDY# and STOP# may be asserted only (immediately) after the address phase.

2.3 Symbolic Timing Diagrams: An introduction by example

In this section, we show how to formulate some of the requirements stated in the last section by a corresponding set of STD–diagrams. At the same time, we will explain informally how STD–diagrams are constructed, and how their semantics is obtained using concepts from automata–theory.

2.3.1 Requirement T1

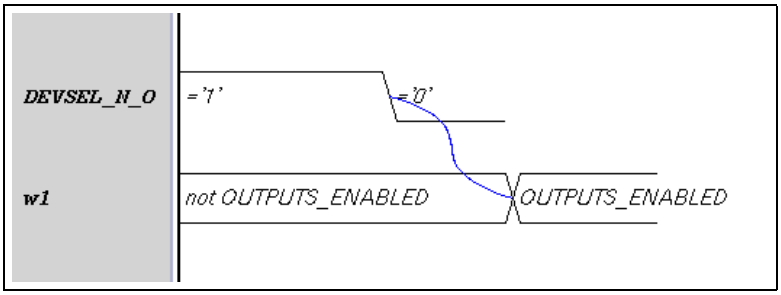
Textual formulation(operating rule 17): *DEVSEL# must be asserted with or prior to the edge at which the target enables its outputs (TRDY#, STOP#, or (on a read) AD lines).*

According to requirement T1, none of the considered outputs may be enabled before DEVSEL# is asserted, as expressed by diagram *T_not_oe_before_de_10* shown in figure 2.2.

We will explain the following items of the diagram *T_not_oe_before_de_10* :

- Graphical syntax: Symbolic waveforms and constraints
- Concrete diagram as template instances

Figure 2.2: Diagram *T_not_oe_before_de_10* .



- Semantics in terms of symbolic automata

At first sight, the diagram looks very much like a "classical" timing diagram, except for the fact that no real–time axis is shown. The diagram has two waveforms, named *DEVSEL_N_O* and *w1*, respectively. The first waveform name coincides with the name of a signal <sig>. This is a special case in STD: it means, that the conditions shown as annotation of the associated waveform must have the form '= <expr>' , which is completed to the expression: <sig> = <expr>.

The second waveform named *w1* shows the general case of a waveform. The name of the waveform has the only purpose to uniquely identify the associated waveform. The waveform consists of two regions, labelled

waveform	*region assertion*
w1.0	not OUTPUTS_ENABLED
w1.1	OUTPUTS_ENABLED

where OUTPUTS_ENABLED is an abbreviation for a Boolean expression defined in the global scope of the specification. The concrete expression depends on implementation details and does not matter here.

In the special case that a waveform describes a two–valued signal (as is the case for signal *DEVSEL_N_O*), then the value = '1' respectively = '0' can be graphically emphasized by corresponding high and low–levels of the waveform. Note, however, that this is just a display option for better readability, and has no semantic meaning. The default graphical shape of a waveform region is the lozenge–shape, as shown for example on waveform *w1* in figure 2.2.

The points separating waveform regions — graphically displayed by falling and/or rising lines — are called (symbolic) *events* and represent possible changes of the component interface state. A symbolic event has the *duration* of one model step.

Time can be imagined to proceed as a cut through the waveforms, progressing from left to right. When the timeline cuts the waveforms in regions labelled e.g. by Boolean expressions <bool–expr–1> and <bool–expr–2>, then this means that the actual component interface state must satisfy the conjunction of all expressions cut by the time–line (e.g.: <bool–expr–1> *and* <bool–expr–2>).

The progress of the timeline may be restricted by *constraints*. A constraint originates from one event *to the right* (the *source* of the constraint) and enters into another event *from the left* (the *target* of the constraint).

The diagram *T_not_oe_before_de_10* has only one constraint, which originates from the (only) event on the waveform associated with signal *DEVSEL_N_O* and ends at the (only) event on the waveform named *w1*. The shape of this constraint is a curved line, which denotes a so–called *precedence constraint*: The target event is required *not to happen before* the source event of the constraint while the timeline moves forward. The automaton representation of this diagram given later in this section will make this concept precise.

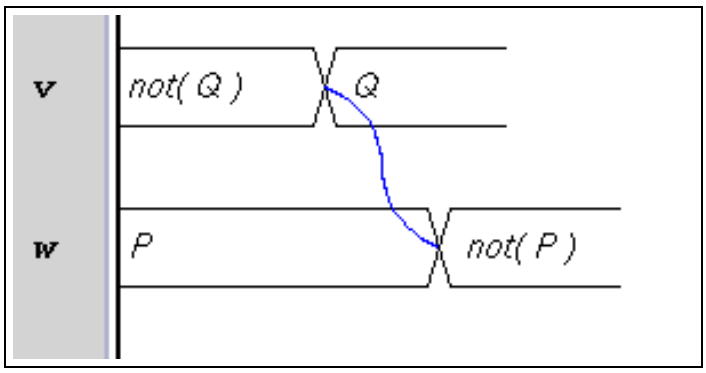
It is often useful to construct diagrams as instances of (generic) templates. For example, the diagram *T_not_oe_before_de_10* can be derived from the diagram–template *TL_P_and_not_Q_unless_Q* shown in figure 2.3, using the mapping:

Parameter	*Mapping*
Q :	Q $\longmapsto$ *DEVSEL_N_O* = '0'
P :	P $\longmapsto$ not(OUTPUTS_ENABLED)

The semantics of this diagram is illustrated in figure 2.4. Model execution is performed in an infinite sequence of steps, starting from step 0. A so–called *activation* of diagram *TL_P_and_not_Q_unless_Q* occurs, whenever the activation–condition of that diagram is met. The activation condition is defined as conjunction of the first predicates taken from all waveforms of the diagram. Thus, the activation condition of diagram *TL_P_and_not_Q_unless_Q* is *(not(Q) and P)*.

When activation happens in some step i (which may happen more than

Figure 2.3: Diagram–template *TL_P_and_not_Q_unless_Q*.



once, even infinitely often during a model–run), then the diagram is *matched* from step i on.

Matching is defined by the following rule: Whenever the state of a component interface changes, then the effect of this change on each currently active diagram of a specification is evaluated.

There are two possibilities (with respect to a particular diagram of the specification):

- The new valuation still satisfies the conditions pointed to by the actual matching–timeline
- In one or more waveforms of the diagram, the event lying ahead of the actual matching–timeline is matched, which means that the timeline is advanced. However, matching must conform to the constraints of the diagram. For instance, the target event of the constraint of diagram *TL_P_and_not_Q_unless_Q* must not be matched before the source event.

In the following, the semantics will be explained in an automata–theoretic setting (which will be familiar to a reader with theoretical background).

In order to define the semantics of the matching process, an automaton is constructed which takes each possible run of the matching process into account. This is illustrated in figure 2.5 , which shows in the form of a transition system how the timeline advances along the diagram during matching.

The automaton shown in figure 2.6 — called the *unwinding structure* of the diagram – is constructed from the transition system shown in figure 2.5.

Figure 2.4: Activation scenario of diagram–template *TL_P_and_not_Q_unless_Q*. Ignore what happens beyond step i ; this will be explained later in this chapter.

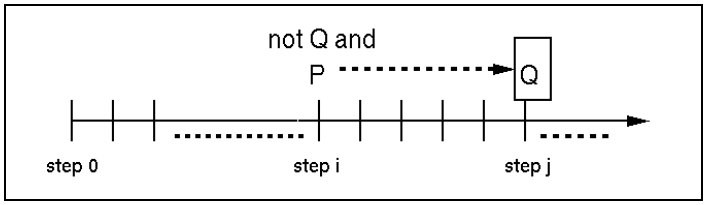


Figure 2.5: Advancement of timeline during matching process.

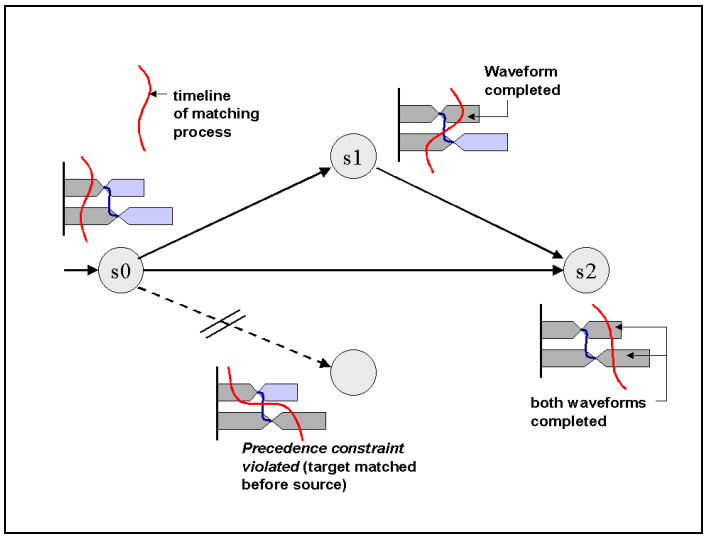
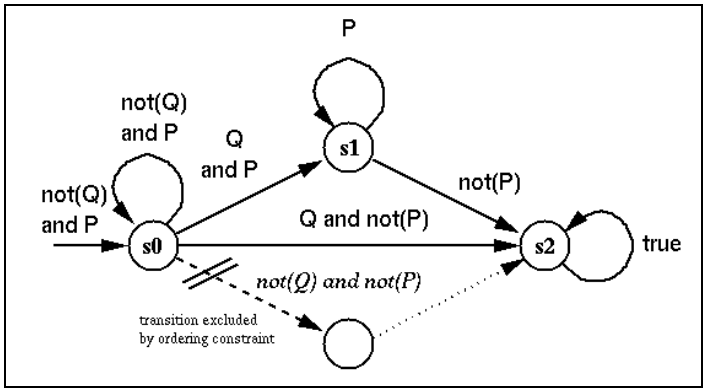


Figure 2.6: Unwinding structure obtained from diagram *TL_P_and_not_Q_unless_Q* .



The transition labels are Boolean expressions constructed from the expression annotations of the waveform regions, which are cut by the timeline *after* the matching step has happened.

The loop label of a state `s` is always either identical to the label of the transition which leads into `s` , or a weaker condition. The latter exception is due to the fact that the final expressions of waveform which have been completely matched, are no longer considered to be restricting the running behaviour. Thus, the end of a waveform is (by default) labelled with the Boolean expression *true* , although this is not displayed. In chapter 6, this is explained in detail.

This automaton accepts — by definition – all behaviour suffixes starting from the next step after activation of a diagram has happened. It follows from the construction that the transition relation of the unwinding structure automaton always forms a directed acyclic graph (DAG) with self–loops. This property is the key for describing later the semantics in terms of temporal logic.

In order to explore the "language" (set of behaviours) accepted by the unwinding automaton, the following sequence of semantics–preserving transformations is applied.

First, the DAG can be expanded into a tree; the construction is obvious, the result is shown in figure 2.7.

Second, the upper branch of the automaton shown in figure 2.7 can be "tail–optimized", which produces the optimized version of the automaton

Figure 2.7: Unwinding structure expanded into tree.

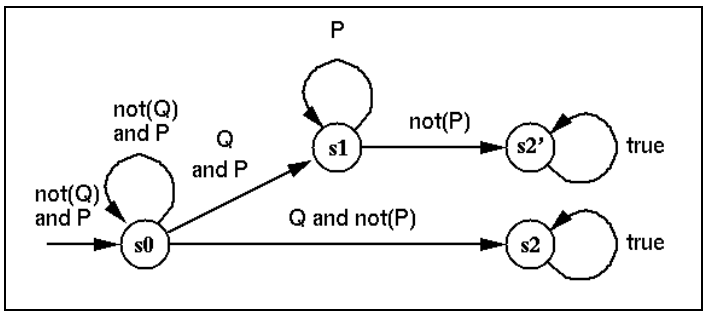
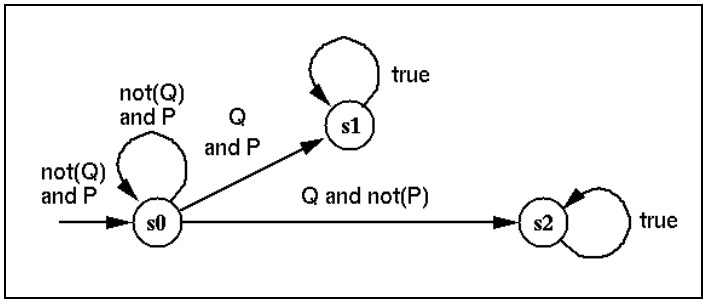


Figure 2.8: Unwinding structure expanded into tree, optimized.



shown in figure 2.8 .

After this optimization step, further simplification becomes possible. The result is shown in the next figure 2.9; note that the condition for a transition from state s0 to s2 is equivalent to Q.

We now reconsider figure 2.4 (see figure 2.10); it remains to explain what happens beyond step i . From step $i+1$ on, the automaton shown in figure 2.9 can be considered to "watch" the ongoing behaviour. The automaton allows two further continuations:

1. *All* steps following step i satisfy condition *(not Q and P)* (control loops in state *s0*)

2. After a finite number of steps following step i , which all satisfy condition

Figure 2.9: Unwinding structure expanded into tree, further simplified.

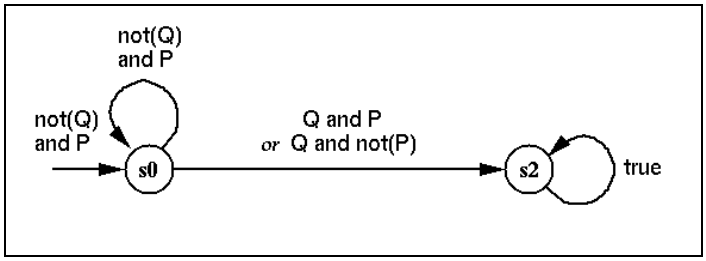
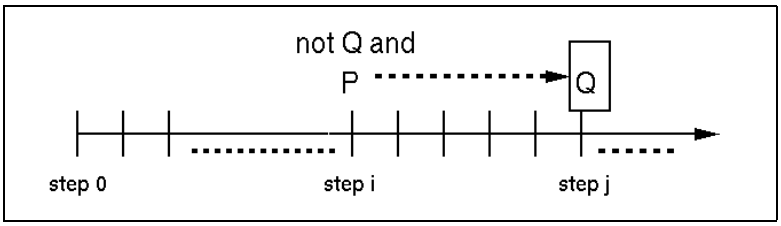


Figure 2.10: Activation scenario and semantics of diagram–template *TL_P_and_not_Q_unless_Q*.



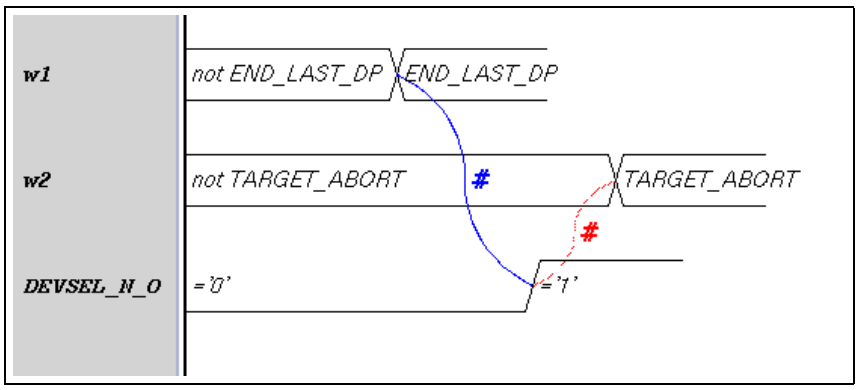
(not Q and P), a step j follows which satisfies condition *Q* . In this moment, control in the unwinding structure advances from state *s0* to state *s2*, which accepts whatever behaviour follows. The activation of the diagram which took place in step i is then satisfied. Note, however, that there may be other activations of the same diagram later on.

2.3.2 Requirement T2

Textual formulation (operating rule 18): *Once DEVSEL# has been asserted, it cannot be de–asserted until the last data phase has completed, except to signal target abort.*

According to requirement T2, once asserted (= '0'), DEVSEL# can only be de–asserted after the end of the last data phase. The exception (target abort) of the requirement is implemented in diagram *T_not_de_01_until_eldp* in waveform *w2* using a so–called *weak* constraint (depicted by a dashed curved

Figure 2.11: Diagram *T_not_de_01_until_eldp*.



line, cf. figure 2.11 and the following explanation).

If DEVSEL# is de–asserted to signal target abort (i.e. at the same time where TRDY# is de–asserted and STOP# is asserted), then the diagram is deactivated.

Diagram *T_not_de_01_until_eldp* consists of three waveforms, consisting of one symbolic event each, and two constraints. The two constraints have different colour and line–type (black/continous versus light grey/stippled). The "normal" constraint type — referred to as *strong* constraint – is drawn in black/continous and expresses a *requirement* with respect to the matching process.

By contrast, the other type — referred to as *weak* constraint, drawn in light grey/stippled line style — expresses an *expectation* with respect to the matching process. This means that the requirements imposed on the matching process by constraints are expressed under the *expectation* that the matching process satisfies the weak constraints.

In other words, if the matching process does violate a weak constraint, then this has the effect of *preemption* on the matching process: The matching process terminates and the diagram accepts (!) the behaviour detected up to the step where the preemption occurred.

In summary, the effect of an implication is obtained: For each step of the matching process, the step either violates (at least) one weak constraint (the "premise"), or it does not violate any weak constraints, and then it must not violate any strong constraints in this step (the "conclusion").

Stated in one sentence, the requirement on each step can be described in the form:

> If no weak constraints are violated,
> then no strong constraint must be violated.

Looking again at the constraint shapes of figure 2.11, it can be seen that both have the form of precedence constraints, with an additional label # attached. This constraint type is called *precedence + conflict*, which is — for strong constraints — a stronger requirement than for normal precedence. It requires in addition, that source and destination event of the constraint must not be matched in the same step. This semantics applies equally for strong and weak constraints.

2.3.3 Requirement T3

Textual formulation (operating rule 12b): *Once asserted, STOP# must remain asserted until FRAME# is de–asserted, whereupon STOP# must be de–asserted.*

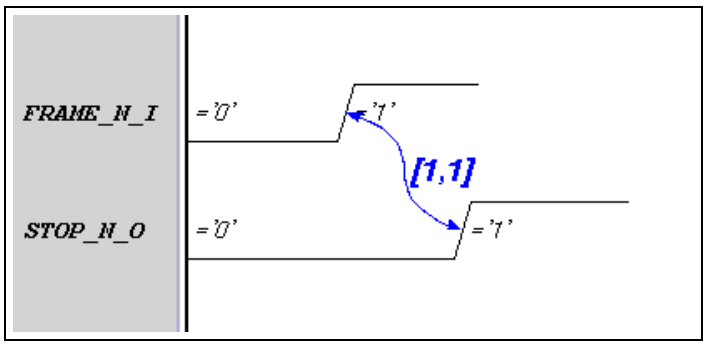
In this case it is possible to capture both requirements of the above rule in one diagram, named *T_st_01_after_fr_01* (figure 2.12):

First, STOP# must not be de–asserted before FRAME# is de–asserted, and second, STOP# must be de–asserted immediately after the de–assertion of FRAME#. So whenever both FRAME# and STOP# are asserted, the first requirement is the FRAME# de–assertion. In the following cycle STOP# must be de–asserted as well.

Here the constraint has the shape of a *distance measure* (arrow–heads both at the source an the target of the constraint), with an interval annotation defining the required distance in terms of model steps. In fact, this type of constraint is not part of STD, but has been introduced in an extension of STD named STDx [31].

Since we are assuming here the discrete step semantics of a synchronous model, in standard STD the lower bound $[1, ...$ of the interval could be expressed by a *precedence+conflict* constraint. The upper bound $..., 1]$ cannot be expressed directly using constraints in STD. We will show in chapter 5, how this part of the requirement can be expressed in the related formalism named LSTD using an extension of the concept of symbolic waveforms.

Figure 2.12: Diagram *T_st_01_after_fr_01*.



2.3.4 Requirement T4

Textual formulation (operating rule 12c): *Once a target has asserted TRDY# or STOP# it can not change DEVSEL#, TRDY# or STOP# until the current data phase completes.*

This rule includes five different requirements regarding which signal is asserted (TRDY# or STOP#), which signal must not change (DEVSEL#, TRDY# or STOP#) and which type of change is considered (assertion or de–assertion). As an example, we consider diagram *T_not_de_01_until_ecdp* (figure 2.13), which states that if TRDY# or STOP# is asserted while DEVSEL# is asserted, DEVSEL# must not be de–asserted until the end of the last data phase.

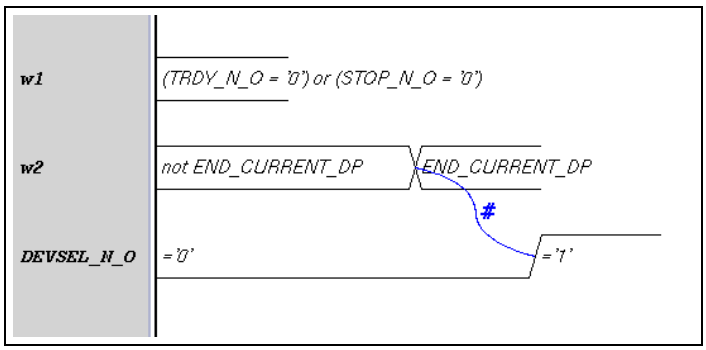
This diagram consists of two "proper" waveforms, and one "stubbed" waveform, where only the first condition is present (*(TRDY_N_O = '0') or (STOP_N_O = '1')*). The stubbed waveform expresses an *activation context*: Diagram activation shall take place only in situations, where the context condition (*(TRDY_N_O = '0') or (STOP_N_O = '1')*) holds.

The notion of an activation context can be implemented as further attribute of a diagram, which has been done in the extension STDx.

2.4 Carrying out verification in a tool environment

The most important requirement about the conception of the formalism STD was the ability to carry out fully–automatic formal verification of synchronous models against requirements formalized by STD diagrams.

Figure 2.13: Diagram *T_not_de_01_until_ecdp*.



By the time of the first publication of the basic concepts of STD ([32]), a technique for verification called *model–checking* was already available in several implementations. A verification environment supporting model–checking of models generated from VHDL– or Statemate–designs (to name just two examples) can have the coarse structure depicted in figure 2.14. In the figure, the data–flow of a verification environment for Statemate^{TM} designs is shown.

The following data–formats play a key role in this verification environment:

- STD — An ASCII–file format representation for STD–specifications. These representations are created and modified in a design–capture tool (graphical editor and manager) for STD. When a version of the specification has been completed, then this version is checked into a verification database.
- Component Profiles — As explained earlier in this chapter, verification is usually carried out not on a full–design, but on a sub–tree of the design, called a (component) *profile* in the Statemate terminology. Designs are created and profiles are defined in the Statemate–design environment.
- Temporal Logic — STD–specification are translated into equivalent temporal logic representations, which can be used as input format for a (symbolic) model–checker such as the VIS model–checker. The definition of this translation is a central part of this book.
- Finite State Machine — A symbolic representation of the finite state machine defining the chosen semantics for the design (depending on the

modelling language, there are options like: choosing a certain clocking scheme, step– vs. superstep semantics etc).

- Error Path — A representation of an error–path can be translated back in the form of a timing diagram. The big advantage of a visual formalism such as STD lies in the fact that the error diagnosis format is then very close to the specification format.

The verification environment sketched in figure 2.14 is described in detail in the thesis [6]; it has evolved from a close collaboration between the company i–Logix and the research centre OFFIS. The verification base engine is the VIS–model–checker [34].

Verification of VHDL designs. Another existing verification for VHDL–designs called CVE has been extended to be used together with the STD–specification formalism. The CVE–verification environment was originally developed by Siemens [10]; later on, it was extended and marketed by the company Abstract–Hardware under the product name CheckOff.

During a cooperation of several years with the research centre OFFIS, this company also provided a graphical design capture tool for STD–specification development, which consists of the STD–database–manager (STD–manager, for short), and the STD–editor, which allows to create STD–diagrams of the kind shown in the examples in this chapter.

To illustrate these activities, we will demonstrate the CheckOff interface in a few snapshots.

The main–window of the CheckOff–system is depicted in figure 2.15. The work–area of the main window depicts, what happens during a model–checking proof. First, a model is generated (automatically) from a given VHDL–design. Similar to the Statemate–environment, a sub–tree of the design is usually selected for compilation. Here, the example introduced at the beginning of this chapter is shown, where the composition consisting of the PCI–sequencer and the PCI–backend component was compiled into a single finite–state–machine (*slave_ctrl_seq_arch_1*).

A *modification step* was applied to remove unnecessary outputs from the model, resulting in a new model named *slave_ctrl_seq_arch*. Other modifications could be applied in a sequence, e.g. setting input pins to constant values.

Right to the icon of the generated finite–state–machine of the implementation, another icon type can be seen (named *slave_ctrl_seq*), which represents a temporal–logic formula, which was generated from one selected diagram of the STD–specification.

Figure 2.14: Architecture of verification environment based on model–checking.

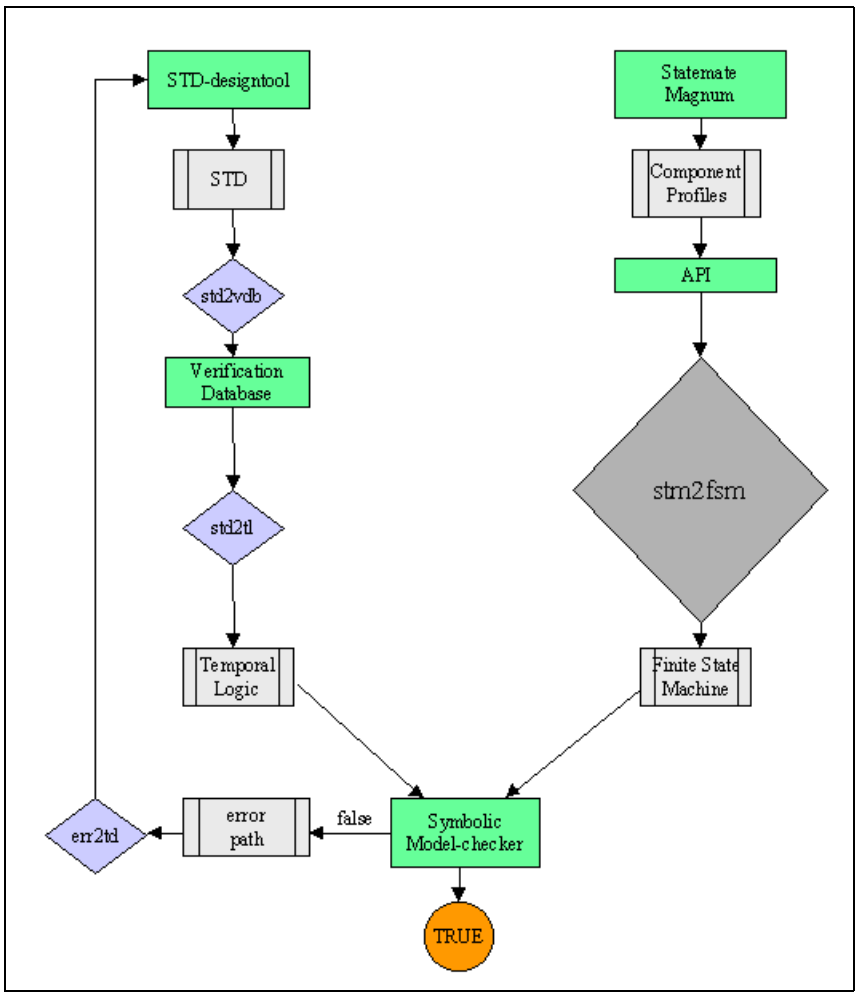


Figure 2.15: Main window of the CheckOff–toolset.

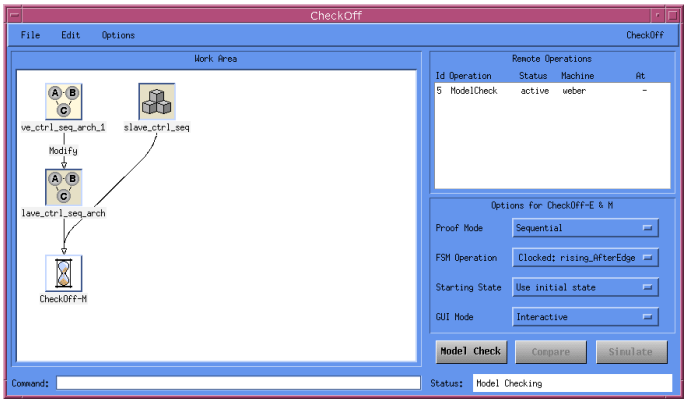


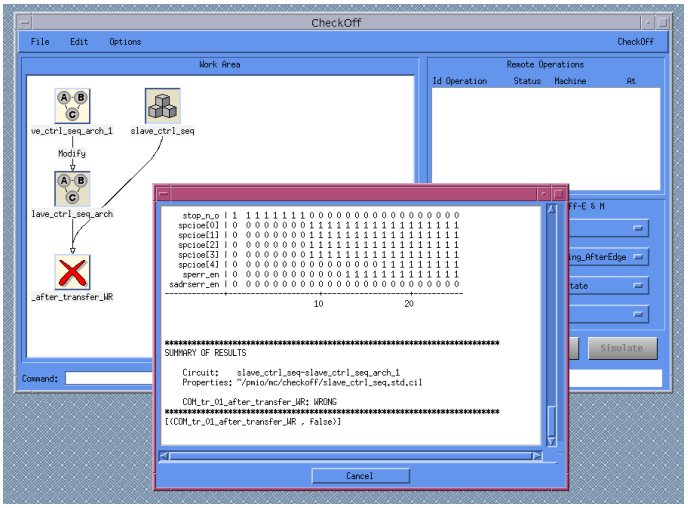
Figure 2.15 just shows the situation, where the model of the (selected part of the) VHDL–design and the temporal logic formula generated from the selected STD–specification clause are checked for conformance. This is done by the verification–module CheckOff–M (for Model–checking). While the verification process is running, an hourglass is shown.

To the right of the work area, further buttons for verification process customization can be seen. The *Proof Mode* can be changed from *Sequential* (model–checking mode) to *equivalence–checking mode* (checking that two implementations are either structurally or — at least – input/output–equivalent). Also, semantic modes according to different clocking schemes can be selected. Finally, the starting state (step 0) of the model can be defined to be either the initial state (based on the definition of the VHDL–language), or the state which the design will reach after a reset has occurred.

The result of a model–checking run is either *true* — which means that the implementation model satisfies the requirements specification — or *false* , which means that at least one system run exists, which violates the selected specification clause.

The result *false* is always useful (assuming that the specification logic is correct): It demonstrates that for a particular sequence of input stimuli (one which may have been overlooked during testing), the component will respond in a way which violates the requirement. A great advantage of the model–checking approach is that it is possible (as far as complexity permits it) to

Figure 2.16: Error–diagnosis explains the violation of a requirement.



automatically construct a counter–example, which demonstrates how the violation occurred.

The CheckOff system provides two ways of error diagnosis. First, there is a low–level debugging interface, which is shown in figure 2.16.

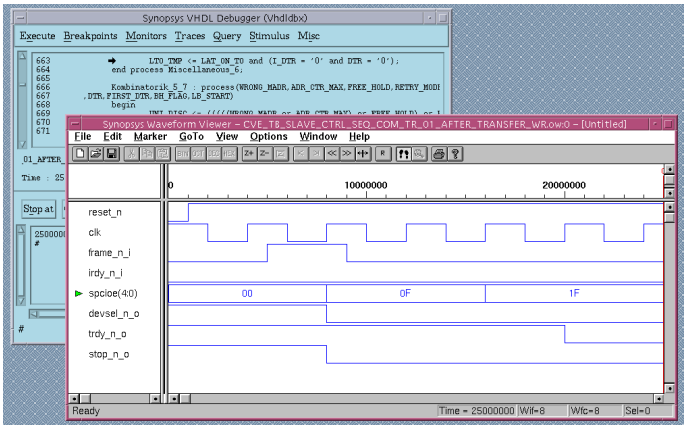
The error–path is shown as a sequence of values assigned to the component interface signals in a sequence of model steps, corresponding to the chosen clocking scheme.

While this is perfectly adequate for a verification engineer, it is also possible to construct a test–environment in order to be able to demonstrate a particular erroneous behaviour. The system automates the construction of this test–environment, so that it can be executed immediately using a standard commercial simulator.

An example is shown in figure 2.17. Note that it is possible to step through the code, so that the position where the error occurred can be precisely identified.

A similar option exists for the verification environment built around Statemate. It is possible to translate an error–path into a so–called *simulation control program* (SCP), which can be used with the Statemate simulator, to drive the model precisely into the state where the error occurred.

Figure 2.17: Simulating an error–path found by model–checking.



2.5 Summary

In this chapter, the basic motivation for the introduction of a graphical specification formalism aiming at formal verification of industrial designs was laid out.

First, examples were shown how requirements can be translated into corresponding STD–diagrams. It was explained how the semantics of individual STD–diagrams is constructed, based on the concept of diagram activation and the unwinding–structure (automaton) as acceptor of a behaviour continuing from the point of diagram activation.

Second, the principle software architecture for a verification environment based on model–checking against STD–specifications has been sketched, corresponding to an existing verification environment which has been built around the Statemate^{TM} design environment.

It has been argued that the model–checking approach can be used with a similar software architecture for different design languages, including VHDL, Verilog, and Statemate. The great advantage of the model–checking approach is given by the fact that it is possible to construct automatically testing environments in order to analyze a particular behaviour which violates a given diagram (specification clause).

Up to now, verification environments supporting STD for requirement specification have been built for VHDL and Statemate.

Three questions arising immediately from the suggested approach are: Have we written enough properties? Are the informally stated requirements correctly translated into diagrams? Are the requirements consistent?

These questions can in principle be answered based on an evaluation of the semantics of the formalized specification, e.g. by synthesizing (C–code) controllers from specifications and testing the behaviour of the code. This approach has been considered by [23] (which includes as example a robotics–application case study) and recently by [25].

Chapter 3

Theoretical foundation of specification

In this chapter, the prerequisites for a formal treatment of the semantics of STD–specifications are introduced.

The following items are discussed:

- Assertion language,
- Boolean expressions,
- Symbolic Automata (SA),
- runs and validity,
- specification of properties (examples),
- partially ordered SA,
- temporal logic and sub–logics thereof.

Note that the exposition style in this chapter is on an elementary level, so that it should be understandable by readers with basic logic background (which can be assumed e.g. for a hardware designer). It will be sufficient for the logic expert to read over this chapter fast in order to recognize the definitions, theorems and citations.

3.1 Assertion Language

The connection between an abstract specification and a concrete implementation model is established using the sub–language of Boolean–expressions.

The sub–language of Boolean expressions can be used as an assertion language. For instance, the language VHDL has a statement assert(<VHDL–Boolean–expression>) which is checked by a simulator during program execution (simulation). For example, suppose that two signals *Grant1* and *Grant2* must never be high at the same time. In order to check this requirement, it is possible to insert a statement: assert(*not*(*Grant1* = '1' *and* *Grant2* = '1')) within the code at all critical instances.

The use of an assertion language is an integral part of all formal program verification techniques. The classical method of (sequential) program verification established by Hoare in the late 60's [19] uses *correctness formulas* (CF's) of the form

$$\{P\}\ S\ \{Q\}$$

where P, Q are assertions (Boolean expressions) constructed from the program variables using some set of operations, and S is a (sequential) program statement (e.g. a variable assignment).

For instance, the correctness formula

$$\{\ \mathsf{n} > 0\}\ \mathsf{n} := \mathsf{n} - 1\ ;\ \ \{\ \mathsf{n} \geq 0\}$$

claims that for any program state where the natural–type variable n is positive, execution of the statement $S \equiv \mathsf{n} := \mathsf{n} - 1$; yields a program state where $\mathsf{n} \geq 0$. Note that the assertion $\mathsf{n} > 0$ abstracts from (the values of) any program variable other than n.

The concrete implementation language, called $I\mathcal{L}$, and its concrete sub–language of Boolean expressions does not matter here; all that is of interest is the fact, that a given Boolean expression bexpr defines a *set of solutions* (mappings of a specified set of variables to concrete values).

The general expression–language $E\mathcal{L}$ of the implementation language $I\mathcal{L}$ is a *family* of languages

$$E\mathcal{L} \equiv_{Def} \bigcup E\mathcal{L}^{\tau},$$

where $E\mathcal{L}^{\tau}$ is the language of expressions of type τ.

The assertion language $A\mathcal{L}$ is a particular expression–language, where ex-

pressions are of the type *Boolean*:

$$A\mathcal{L} \equiv_{Def} E\mathcal{L}^{Boolean} \quad .$$

A program semantics assigns to each expression $E^\tau \in E\mathcal{L}^\tau$ constructed from a set of (typed) variables

$$\mathcal{V} \equiv_{Def} \bigcup \mathcal{V}^\tau$$

a value

$$[\![E^\tau]\!]\rho \in \mathcal{D}^\tau$$

where ρ is a (type–consistent) *valuation* of the variables in $\mathcal{V}$, i.e. a mapping

$$\rho \in Val(\mathcal{V}) \ : \ v^\tau \in \mathcal{V}^\tau \mapsto d_v^\tau \in \mathcal{D}^\tau \quad ,$$

where $\mathcal{D}^\tau$ is the domain of values of type τ, and $Val(\mathcal{V})$ is the domain of valuations of $\mathcal{V}$.

E.g., the domain of Boolean constants is

$$\mathcal{D}^{Boolean} \equiv_{Def} \{\mathbf{true}, \mathbf{false}\} \quad .$$

The precise (language dependent) definition of the expression semantics $[\![.]\!]\rho$ is not important for our exposition and is assumed to be given [1]. The main point is that the semantics of a Boolean–type expression $E^{Boolean}$ can be used to define a set of valuations $L \subseteq Val(\mathcal{V})$ of the variables in $\mathcal{V}$, which contains all valuations satisfying $E^{Boolean}$ as follows:

$$L \equiv_{Def} \{\rho \subseteq Val(\mathcal{V}) \mid [\![E^{Boolean}]\!]\rho = \mathbf{true}\}.$$

On the level of logic, we use the domain of truth values

$$\mathcal{D}^{truthval} =_{Def} \{\mathrm{TRUE}, \mathrm{FALSE}\}$$

which should not be confused with the domain $\mathcal{D}^{Boolean}$ of values of Boolean expressions.

[1] We deliberately avoid to go into the details of the definition of the assertion language. See appendix B for additional notes on this topic.

Next, we define the language of Boolean expressions.

Definition 3.1 (Classical logic expressions based on an assertion language) *Assume an assertion language $A\mathcal{L}$, and a set $\mathcal{V}$ of (typed) variables. Then the language of classical logic expressions based on $A\mathcal{L}$ is defined by the following grammar with the production sets (0)–(1):*

$$\begin{array}{rcll} \phi & \longrightarrow & \langle\!|E|\!\rangle \mid (\phi_1) & (0) \\ & \mid & \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 & (1) \end{array}$$

where E is a Boolean expression of the assertion language $A\mathcal{L}$ over $\mathcal{V}$.

The set of formulas which can be derived from the productions (0) — (1) is referred to as $Bool_{A\mathcal{L}}$, called the Boolean fragment of the assertion language $A\mathcal{L}$. Note: The logical constants TRUE and FALSE are syntactically represented as $\langle\!|\mathbf{true}|\!\rangle$ and $\langle\!|\mathbf{false}|\!\rangle$, respectively. In the context of temporal logic we will later sometimes write β instead of ϕ to emphasize the fact that ϕ is a Boolean formula.

For a formula $\phi \in Bool_{A\mathcal{L}}$, state $\rho \in Val(V)$, we define the validity of a state with respect to the formula, denoted by $\rho \models \phi$.

Definition 3.2 (Validity of a state w.r.t. a formula) *The validity of a state $\rho \in Val(V)$ w.r.t. a formula $\phi \in Bool_{A\mathcal{L}}$ is defined by induction on the structure of ϕ.*

$$\begin{array}{ccc} \rho \models \langle\!|E|\!\rangle & \textbf{iff} & [\![E]\!]\rho \equiv \mathbf{true} \\ \rho \models \phi_1 \wedge \phi_2 & \textbf{iff} & \rho \models \phi_1 \text{ and } \rho \models \phi_2 \\ \rho \models \phi_1 \vee \phi_2 & \textbf{iff} & \rho \models \phi_1 \text{ or } \rho \models \phi_2 \\ \rho \models \neg\phi_1 & \textbf{iff} & \text{not } \rho \models \phi_1 \\ \rho \models (\phi_1) & \textbf{iff} & \rho \models \phi_1 \end{array}$$

A formula ϕ is called a tautology, denoted

$$\models \phi$$

if $\rho \models \phi$ for all states $\rho \in Val(V)$.

3.2 Symbolic Automata

Symbolic Automata are the basis of the semantics definition of STD. The notion of symbolic automata is derived from (non–deterministic) Büchi–automata (NBA, cf. [7] and [36]), extended to allow expressions as transition labels. Correspondingly, the semantics of acceptance has to take into account the notion of assertion satisfaction.

In the following, we consider an assertion language $A\mathcal{L}$ as given.

We will introduce symbolic automata as a formalism, which can be used to specify *qualitive* timing requirements of a system behavior. The formalism is operational in nature and is adequate for the formulation of "abstract" views about a system behavior.

3.2.1 Basic definition

We introduce the definition of symbolic automata without prior motivation; it is close to the well known concept of (nondeterministic) *Büchi automata*.

Definition 3.3 (Symbolic Automaton) *Assume an assertion language $A\mathcal{L}$, and a set V of variables.*

A symbolic automaton (SA) $\mathcal{A}$ over V is a structure

$$\mathcal{A} \ : \ (V, Locs, Edges, L_0, F)$$

with the following components:

- *$V \subseteq \mathcal{V}$ is a finite set of variables,*
- *Locs is a finite set of locations, and*
- *Edges is a finite set of edges. Edges are triples*

$$(\ell, \phi, \ell')$$

  where ℓ, ℓ' are locations, and $\phi \in Bool_{A\mathcal{L}}$ is a formula over V,

- *$L_0 \subseteq Locs$ is the set of initial locations;*
- *$F \subseteq Locs$ is a designated set of acceptance states (whose semantics will be defined below).*

3.2.2 Computations

The formulas of $Bool_{A\mathcal{L}}$ introduced in definition 3.1 can be used to make assertions about the state ρ of a non–terminating (reactive) program at any particular moment in time.

We consider a notion of time which is adequate to describe discrete *synchronous* systems.

The time domain is denoted by *Time*; each element $t \in Time$ is called a *moment*. For models of synchronous systems, a notion of time is appropriate which is isomorphic to the *set of natural numbers*, i.e.

$$Time \equiv_{Def} N_0 \quad .$$

This definition implies (1) that there is a *first moment* 0 in time, (2) time advances in *discrete* steps, (3) time is infinite (there is no "last" moment), and (4) the moments in time are linearly ordered. As a consequence, it is possible to argument by *induction* over the time domain.

Definition 3.4 (Computation sequence) *Given a set of variables V, a computation σ over V (of a system) is a mapping*

$$\sigma \in Comp(V) \ : \ t \in Time \mapsto \rho_t \in Val(V) \quad ,$$

where the set of all computations over V is denoted as $Comp(V)$.

3.2.3 Runs and notion of acceptance

Deterministic automata can be used to classify computations by judging the effect which it has on the internal state of the automaton.

For the definition of the semantics of *nondeterministic* automata, there is usually more than one possible effect on the internal state transitions. In this case, *all possible effects* are taken into account.

Definition 3.5 (Semantics of Symbolic Automaton) *Assume an assertion language $A\mathcal{L}$, a set $\mathcal{V}$ of variables, and a symbolic automaton $\mathcal{A}$ over $V \subseteq \mathcal{V}$.*

Given a computation sequence σ over V, a run $\sigma\ell$ of $\mathcal{A}$ over $\sigma \equiv_{Def} (\rho_i)_{i\geq 0}$ is an infinite sequence of locations of $\mathcal{A}$:

$$\sigma\ell \ : \ i \in N_0 \mapsto \ell_i \in Locs$$

satisfying the following conditions (1) and (2):

$$\ell_0 \in L_0 \qquad (1)$$
$$\forall i \geq 0 \ : \ \exists \phi \ . \ (\ell_i, \phi, \ell_{i+1}) \in \mathit{Edges} \wedge \rho_i \models \phi \qquad (2)$$
$$\{i \mid \ell_i \in F\} \textit{ is an infinite set (of } F\textit{–locations)} \qquad (3)$$

$\sigma\ell$ is called an accepting run, if condition (3) is also satisfied.

The semantics (language) $L(\mathcal{A})$ of SA $\mathcal{A}$ is the set of computation sequences σ over V for which one accepting run over σ exists.

3.2.4 Properties of Symbolic Automata

This section lists a key property of SA which is referenced often in later chapters. The property is called the *statement of monotonicity*: "Relaxing" the label of a transition (i.e. replacing a formula ϕ by a weaker formula ϕ') yields a new automaton, which accepts at least all runs of the original SA.

Lemma 3.1 (Monotonicity of SA) *Assume an assertion language $A\mathcal{L}$, a set $\mathcal{V}$ of variables, and a symbolic automaton $\mathcal{A} = (V, \mathit{Locs}, \mathit{Edges}, L_0, F)$ over $V \subseteq \mathcal{V}$.*

Assume further another SA $\mathcal{A}' = (V, \mathit{Locs}, \mathit{Edges}', L_0, F)$ over V, which is identical to $\mathcal{A}$ except for one edge $e \in \mathit{Edges}$, which becomes e' in $\mathcal{A}'$:

$$e \equiv_{Def} (\ell, \phi, \ell') \qquad \textit{in } \mathcal{A}\textit{, and}$$

$$e' \equiv_{Def} (\ell, \phi', \ell') \qquad \textit{in } \mathcal{A}'$$

where ϕ implies ϕ' ($\phi \Rightarrow \phi'$). The set Edges' is defined by

$$\mathit{Edges}' = \mathit{Edges} \backslash \{e\} \cup \{e'\}$$

Then

$$L(\mathcal{A}) \subseteq L(\mathcal{A}') \quad .$$

Note: The notion of formula implication is given in definition 3.13.

3.2.5 Deterministic Symbolic Automata

The understanding of the semantics of a symbolic automaton is simplified if runs are uniquely determined by given computations. This is the case for deterministic automata.

Definition 3.6 (Complete and deterministic Symbolic Automaton) *Assume an assertion language $A\mathcal{L}$, a set $\mathcal{V}$ of variables, and a symbolic automaton $\mathcal{A} = (V, Locs, Edges, L_0, F)$ over $V \subseteq \mathcal{V}$.*

Define for a location $\ell \in Locs$ the set of formulas

$$\Phi_\ell =_{Def} \{\phi \mid \exists \ell' \in Locs \,.\, (\ell, \phi, \ell') \in Edges\}$$

for which an edge to a successor state of ℓ labelled with a formula $\phi \in \Phi_\ell$ exists. $\mathcal{A}$ is called to be deterministic at ℓ iff

$$\forall \phi_1, \phi_2 \in \Phi_\ell \,:\models \phi_1 \rightarrow \neg\phi_2 \quad ; \qquad \textit{(disjoint labeling)}$$

$\mathcal{A}$ is called to be complete at ℓ iff

$$\models \bigvee_{\phi \in \Phi_\ell} \phi \qquad \textit{(exhaustive labeling)}$$

$\mathcal{A}$ is called complete, if it is complete at all locations $\ell \in Locs$, and called deterministic, if it is deterministic at all locations $\ell \in Locs$ and the set L_0 of initial locations contains exactly one location.

The next lemma is a consequence of the deterministic property.

Lemma 3.2 (Unique run of deterministic SA) *Assume that $\mathcal{A} = (V, Locs, Edges, L_0, F)$ is a deterministic and complete SA over V.*

Then $\mathcal{A}$ has for any given computation sequence $\sigma \equiv_{Def} (\rho_i)_{i \geq 0}$ over V exactly one run $\sigma\ell$ (either accepting of not) over σ, which is defined by

$$\begin{array}{lrcl} \sigma\ell =_{Def} (& 0 & \mapsto & \textbf{any } \ell \,.\, \ell \in L_0 \\ & i > 0 & \mapsto & \textbf{any } \ell' \,.\, \exists \phi \,.\, (\sigma\ell(i-1), \phi, \ell') \in Edges \wedge \rho_{i-1} \models \phi) \end{array}$$

Proof of lemma 3.2:

Definedness of $\sigma\ell$. We have to show that the given definition of $\sigma\ell$ over σ defines $\sigma\ell$ as a function, which is the case if the **any** –constructs used in the definition of $\sigma\ell$ define unique values. Since $\mathcal{A}$ is deterministic, $L_0 = \{\ell_0\}$ for some $\ell_0 \in \mathit{Locs}$,

$\hookrightarrow$[def. of **any**] (**any** ℓ . $\ell \in L_0) = \ell_0$.

It remains to show the well–definedness of $\sigma\ell$ for the second definition–clause (case $i > 0$). Consider for $\ell \in \mathit{Locs}$, $\rho \in \mathit{Val}(V)$ the set

$$S_{\ell,\rho} =_{Def} \{\ell' \in \mathit{Locs} \mid \exists\phi \,.\, (\ell, \phi, \ell') \in \mathit{Edges} \wedge \rho \models \phi\} \quad .$$

Our next goal to show that

$$|S_{\ell,\rho}| = 1 \qquad (*)$$

for all $\ell \in \mathit{Locs}$, $\rho \in \mathit{Val}(V)$.

$S_{\ell,\rho} \neq \emptyset$ **.** By premise, $\mathcal{A}$ is (in particular) complete at ℓ.

$\hookrightarrow$[by condition (exhaustive labeling)] $\rho \models \bigvee\limits_{\phi\in\Phi_\ell} \phi$

$\hookrightarrow$[def. of $\bigvee$] $\exists\phi_0 \in \Phi_\ell \,.\, \rho \models \phi_0$

$\hookrightarrow$[def. of Φ_ℓ, $\phi_0 \in \Phi_\ell$] $\exists\ell' \,.\, (\ell, \phi_0, \ell') \in \mathit{Edges}$

$\hookrightarrow$[$\rho \models \phi_0$] $\ell' \in S_{\ell,\rho}$.

$|S_{\ell,\rho}| < 2$ **.** Assume the opposite, i.e.

$$\exists\ell_1, \ell_2 \in S_{\ell,\rho} \,.\, \ell_1 \neq \ell_2 \qquad (**)$$

For $i = 1, 2$: $\ell_i \in S_{\ell,\rho}$

$\hookrightarrow$[def.of $S_{\ell,\rho}$] $\exists\phi_i \in \Phi_\ell \,.\, (\ell, \phi_i, \ell_i) \in \mathit{Edges} \wedge \rho \models \phi_i$ (*)

On the other hand, by condition (disjoint labeling):

$\hookrightarrow$[$\models \phi_1 \rightarrow \neg\phi_2$] $\rho \models \phi_1 \rightarrow \neg\phi_2$,

$\hookrightarrow$[def. of $\rightarrow$, $\neg$] not ($\rho \models \phi_1$ and $\rho \models \phi_2$).

But this contradicts (*), hence assumption (**) must be false.

$\hookrightarrow$[$S_{\ell,\rho} \neq \emptyset$ and $|S_{\ell,\rho}| < 2$] $|S_{\ell,\rho}| = 1$

Consider again the clause:

$$\textbf{any } \ell' \,.\, \exists \phi \,.\, (\sigma\ell(i-1), \phi, \ell') \in \mathit{Edges} \wedge \rho_{i-1} \models \phi)$$

and let $\ell \equiv_{Def} \sigma\ell(i-1)$, $\rho \equiv_{Def} \rho_{i-1}$.

Then

$$S_{\ell,\rho} = \{\ell' \in \mathit{Locs} \mid \exists \phi \,.\, (\ell, \phi, \ell') \in \mathit{Edges} \wedge \rho \models \phi\} = \{\ell'_0\}$$

for some $\ell'_0 \in \mathit{Locs}$.

$\hookrightarrow$[by def.] (**any** $\ell' \,.\, \exists \phi \,.\, (\sigma\ell(i-1), \phi, \ell') \in \mathit{Edges} \wedge \rho_{i-1} \models \phi) = \ell'_0$

It follows that $\sigma\ell$ is well–defined.

Example 3.1 (Specification of properties using SA) *We assume a formula $\phi \in \mathit{Bool}_{A\mathcal{L}}$ over some set of variables V and define:*

$$\mathcal{A}_{\textbf{eventually } \phi} =_{Def} (V, \mathit{Locs}, \mathit{Edges}, L_0, F)$$

The components Locs, Edges, L_0, F are specified in the conventional diagrammatic notation used for graphs and transition systems, with the following definition: (1) Each initial location $\ell \in L_0$ is represented by a arrow without source pointing to it, and (2) each acceptance state $\ell \in F$ is denoted by a double–circle (as opposed to single–circles for non–acceptance states).

Using these conventions, we define $\mathcal{A}_{\textbf{eventually } \phi}$ over V by the diagram shown in figure 3.1. The automaton accepts all computations over V, which contain at least one ϕ–state.

3.3 Partially ordered SA

We will next consider a particular subclass of SA, which is the class of symbolic automata which contain no (non–trivial) cycles.

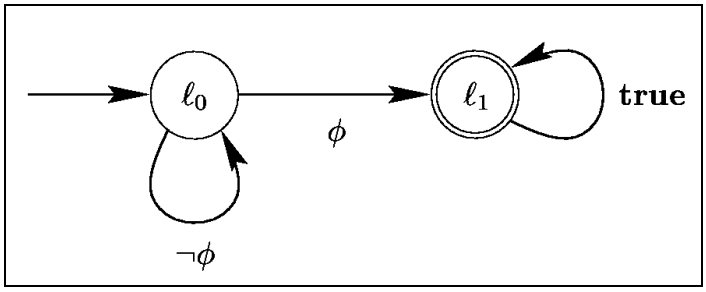
Definition 3.7 (Partially ordered SA) *Assume an assertion language $A\mathcal{L}$, and a set V of variables.*

A partially ordered symbolic automaton (POSA) $\mathcal{A}$ over V is a SA

$$\mathcal{A} \,:\, (V, \mathit{Locs}, \mathit{Edges}^{po}, L_0, F) \qquad \textit{(POSA)}$$

with the following restriction on the set Edges^{po}: Define a binary relation $\rightarrow$ on the set Locs (denoted in infix notation) by

Figure 3.1: Symbolic Automaton $\mathcal{A}_{\textbf{eventually } \phi}$. From this figure on, we will simply write true as edge label instead of $\langle\!|true|\!\rangle$.



$$\ell_1 \rightarrow \ell_2 \textbf{ iff } \exists \phi \,.\, (\ell_1, \phi, \ell_2) \in Edges^{po} \quad .$$

Then the SA $\mathcal{A}$ is called a partially ordered SA, iff the relation $\rightarrow^*$ (the reflexive, transitive closure of $\rightarrow$) is a partial order. In particular, it is required to be anti–symmetric, i.e.:

$$(\ell_1 \rightarrow^* \ell_2) \wedge (\ell_2 \rightarrow^* \ell_1) \quad \Longrightarrow \quad \ell_1 = \ell_2 \quad .$$

For a POSA $\mathcal{A}$, the relation $\rightarrow^*$ is denoted by $\preceq_{\mathcal{A}}$ (or simply by $\preceq$).

Example 3.2 (POSA $\mathcal{A}_{\textbf{eventually } \phi}$) *The SA $\mathcal{A}_{\textbf{eventually } \phi}$ shown in figure 3.1 is a POSA, where*

$$\preceq_{\mathcal{A}_{\textbf{eventually } \phi}} = \{(\ell_0, \ell_0), (\ell_0, \ell_1), (\ell_1, \ell_1)\} \quad .$$

Lemma 3.3 (Runs of a POSA) *Let $\mathcal{A}$ be a POSA over some set V of variables. Then each run $\sigma\ell$ of $\mathcal{A}$ has the form*

$$\sigma\ell = (\ell_0, \ell_1, \ldots, \ell_k, \ell_{k+1}, \ldots)$$

for some $k \geq 0$, where

$$\ell_0 \in L_0 \qquad (1)$$
$$\forall i \,.\, 0 \leq i < k \;:\; \ell_i \preceq \ell_{i+1} \qquad (2)$$
$$\forall i \geq k \;:\; \ell_i = \ell_k \qquad (3)$$

Proof of Lemma 3.3:

Proof of (1),(2) . Let $\sigma\ell \equiv_{Def} (\ell_i)_{i\geq 0}$ be a run of $\mathcal{A}$. Then (1) holds by definition ($\ell_0 \in L_0$). Also by definition:

$$\forall i \geq 0 \;:\; \ell_i \rightarrow \ell_{i+1}$$

$\hookrightarrow [\ell \rightarrow \ell' \implies \ell \preceq \ell']\;\; \forall i \geq 0 \;:\; \ell_i \preceq \ell_{i+1}$

$\hookrightarrow$[(in particular)] (2).

Proof of (3). First, we show that $\sigma\ell$ cannot have non–trivial loops, i.e. for some positions $i_1 < i_3 < i_2$ in $\sigma\ell$: $\ell_{i_1} = \ell_{i_2}$ and $\ell_{i_3} \neq \ell_{i_1}$.

Assume by contrary that this were the case;

$\hookrightarrow [\forall j \,.\, i_1 \leq j < i_2 \;:\; \ell_j \preceq \ell_{j+1}$, transitivity of $\preceq]\;\; \ell_{i_1} \preceq \ell_{i_3}$ and $\ell_{i_3} \preceq \ell_{i_2}$

$\hookrightarrow [\ell_{i_1} = \ell_{i_2},\, \ell_{i_3} \preceq \ell_{i_2} \Rightarrow \ell_{i_3} \preceq \ell_{i_1}$, anti–symmetry of $\preceq]\;\; \ell_{i_3} = \ell_{i_1}$

which contradicts the assumption; hence $\sigma\ell$ cannot have non–trivial loops.

Now assume, by contrary to (3), that for $\sigma\ell$, $\forall k \geq 0 \exists i \geq k \,.\, \ell_i \neq \ell_k$ (*).

Consider a sequence $(\ell'_i)_{i\geq 0}$, defined by:

$$\begin{array}{lll} \ell'_0 & =_{Def} & \ell_0 \\ \ell'_{i+1} & =_{Def} & \ell_k \quad \text{where } k =_{Def} \textbf{ any } j \,.\, j > i \wedge \ell_j \neq \ell'_i \end{array} \qquad (**)$$

By assumption (*), the clause (**) is satisfiable, hence some sequence $\sigma\ell'$ with the required properties exists under the assumption (*).

Let $|Locs| = m \geq 2$ (the case $|Locs| = 1$ is trivial), and consider the set $S_{\sigma\ell',m} =_{Def} \{\ell'_0, \ldots, \ell'_m\}$. Since this set has $m+1$ elements, there must be two indices $j_1 < j_2$, such that $\ell'_{j_1}, \ell'_{j_2} \in S_{\sigma\ell',m}$ and $\ell'_{j_1} = \ell'_{j_2}$. Since by construction $\ell'_{j_1} \neq \ell'_{j_1+1}$, it follows that $j_1 < j_1 + 1 < j_2$. By construction of $\sigma\ell'$, this implies that $\sigma\ell$ would have a non–trivial loop at those positions i_1,i_3,i_2, for

which $\ell'_{j_1} = \ell_{i_1}$, $\ell'_{j_3} = \ell_{i_1+1}$ and $\ell'_{j_2} = \ell_{i_2}$, which is impossible; hence (3) must be true. q.e.d.

Lemma 3.3 implies that for a partially ordered SA $\mathcal{A}$, each run $\sigma\ell$ eventually remains forever at some location $\ell_f \in Locs$, after "climbing up" in $Locs$ w.r.t. the partial order $\preceq$; this final location ℓ_f "decides" whether the run is accepting ($\ell_f \in F_{\mathcal{A}}$) or not ($\ell_f \notin F_{\mathcal{A}}$).

In the following, we will assume that any SA (POSA) is in a normal form, which can be assumed without loss of generality due the next lemma.

Lemma 3.4 (SA–normalization, SA–normal form) *Assume a set V of variables, and some SA $\mathcal{A}$ over V. Then we can find another SA $\mathcal{A}' \equiv_{Def} (V, Locs', Edges', L'_0, F')$ over V, which is language–equivalent to $\mathcal{A}$ (i.e.: $L(\mathcal{A}) = L(\mathcal{A}')$) and satisfies the following normal–form requirements:*

1. *(completeness) $\mathcal{A}'$ is complete (cf. def. 3.6),*
2. *(selfloop–closure) For each location $\ell \in Locs'$, $\ell \to \ell$, i.e. there exists a (selfloop–)edge $(\ell, \phi, \ell) \in Edges'$ for some ϕ, and*
3. *(no–parallel–edges) For all locations $\ell, \ell' \in Locs'$, $\ell \neq \ell'$ holds:*

$$(\ell, \phi_1, \ell') \in Edges' \wedge (\ell, \phi_2, \ell') \in Edges' \quad \Longrightarrow \quad \phi_1 = \phi_2 \quad .$$

A SA which has the three properties stated above is said to be in SA–normal form.

For example, the automaton $\mathcal{A}_{\mathbf{eventually}\ \phi}$ shown in figure 3.1 is in SA–normal form.

3.4 (Linear–time) Temporal Logic

We further assume some assertion language $A\mathcal{L}$ as given.

We next introduce the language of (linear–time) temporal logic on top of a given assertion language.

Definition 3.8 ((linear–time) temporal logic) *Assume an assertion language $A\mathcal{L}$, and a set V of variables. Then the language of linear–time temporal logic based on $A\mathcal{L}$ ($LTL_{A\mathcal{L}}$) is defined by the following grammar with the production sets (0)–(4):*

$$
\begin{array}{rcll}
\phi & \longrightarrow & \langle\!|E|\!\rangle \mid (\phi_1) & \textit{(0)} \\
& \mid & \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 & \textit{(1)} \\
& \mid & \textbf{always}\,\phi_1 \mid \textbf{eventually}\,\phi_1 & \textit{(2)} \\
& \mid & \phi_1\,\textbf{until}\,\phi_2 \mid \phi_1\,\textbf{unless}\,\phi_2 & \textit{(3)} \\
& \mid & \textbf{next}\,\phi_1 & \textit{(4)}
\end{array}
$$

where E is a Boolean expression of the assertion language $A\mathcal{L}$ over V. Without the production (4), the grammar defines the language $LTL_{A\mathcal{L}}$ without "nexttime–operator" (denoted $LTL_{A\mathcal{L}}{}^{st}$).

We allow in addition using the Boolean connectives $\rightarrow$ and $\leftrightarrow$, which are defined as abbreviations:

$$
\begin{array}{lll}
\phi_1 \rightarrow \phi_2 & \equiv_{Def} & (\neg\phi_1 \vee \phi_2) \\
\phi_1 \leftrightarrow \phi_2 & \equiv_{Def} & (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)
\end{array}
$$

The precedences of the operators introduced in definition 3.8 are as follows: The unary operators $\neg$, **always** , **eventually** , **next** bind stronger than the binary operators **until** , **unless** , which bind stronger than $\wedge$; as usual, $\wedge$ binds stronger than $\vee$, which binds stronger than $\rightarrow$, $\leftrightarrow$. All binary operators are left–associative. Parentheses (and) can be used to define the grouping of sub–formulas.

The interesting part of definition 3.8 is the introduction of the temporal operators **always** , **eventually** , **until** , **unless** and **next** in the production sets (2)–(4). We will explain the meaning of these operators together with the formal semantics definition of *LTL*.

3.4.1 Formal semantics of temporal logic

Given a set of variables V, the semantics of the temporal logic language is defined in terms of *validity formulas* of the form

$$\sigma \models_V \phi \quad ,$$

where σ is a computation, ϕ is a temporal logic formula, and the operator $\models_V$ is read "satisfies"; if the set of variables V is clear from the context, we simply write $\models$ instead of $\models_V$.

For the semantics definition, we define as preparation the notion of the *j–suffix* of a computation σ, denoted as $\sigma^{(j)}$:

$$\sigma^{(j)} \ : \ i \in N_0 \mapsto_{def} \sigma(i+j) \qquad \text{(computation–suffix)}$$

Definition 3.9 (Semantics of temporal logic) *Assume an assertion language $A\mathcal{L}$, and a set V of variables. Then the semantics of the temporal logic language $LTL_{A\mathcal{L}}$ is defined by induction on the structure of a formula $\phi \in LTL_{A\mathcal{L}}$.*

For any computation $\sigma \in Comp(V)$, the validity of the clause $\sigma \models \phi$ is defined by induction on the structure of ϕ.

$$\begin{array}{lcll}
\sigma \models & \langle\!|E|\!\rangle & \textbf{iff} & [\![E]\!]\sigma(0) = \textbf{true} \\
\sigma \models & (\phi_1) & \textbf{iff} & \sigma \models \phi_1 \\
\sigma \models & \phi_1 \wedge \phi_2 & \textbf{iff} & \sigma \models \phi_1 \ \textit{and} \ \sigma \models \phi_2 \\
\sigma \models & \phi_1 \vee \phi_2 & \textbf{iff} & \sigma \models \phi_1 \ \textit{or} \ \sigma \models \phi_2 \\
\sigma \models & \neg\phi_1 & \textbf{iff} & \textit{not} \ \sigma \models \phi_1 \\
\sigma \models & \textbf{always}\,\phi_1 & \textbf{iff} & \forall j \geq 0 : \sigma^{(j)} \models \phi_1 \\
\sigma \models & \textbf{eventually}\,\phi_1 & \textbf{iff} & \exists j \geq 0 : \sigma^{(j)} \models \phi_1 \\
\sigma \models & \phi_1\,\textbf{until}\,\phi_2 & \textbf{iff} & \exists j \geq 0 : \sigma^{(j)} \models \phi_2 \ \textit{and} \\
 & & & \forall i : 0 \leq i < j : \sigma^{(i)} \models \phi_1 \\
\sigma \models & \phi_1\,\textbf{unless}\,\phi_2 & \textbf{iff} & \sigma \models \phi_1\,\textbf{until}\,\phi_2 \ \textit{or} \ \sigma \models \textbf{always}\,\phi_1 \\
\sigma \models & \textbf{next}\,\phi_1 & \textbf{iff} & \sigma^{(1)} \models \phi_1 \quad .
\end{array}$$

With definition 3.9 at hand, we will next explain the informal meaning of the temporal logic operators **always** , **eventually** , **until** , **unless** and **next** .

First, **always** ϕ_1 holds for a computation σ, iff ϕ_1 holds *for all moments* in the *Time*–domain of σ; by contrast, **eventually** ϕ_1 holds for σ, iff ϕ_1 holds (at least) for *one moment* of the domain of σ.

Second, ϕ_1 **until** ϕ_2 holds for a computation σ, iff ϕ_2 holds (at least) for *one moment* of the domain of σ *and* ϕ_1 holds *for all preceding moments* of the time–domain of σ (note that the domain *Time* is linearly ordered).

Similarly, ϕ_1 **unless** ϕ_2 holds for a computation σ, iff either ϕ_1 **until** ϕ_2

holds for σ (i.e. in particular ϕ_2 holds (at least) for one moment of the time–domain of σ) *or* ϕ_1 holds *for all moments* of the time–domain of σ.

Third, **next** ϕ_1 holds for a computation σ, iff ϕ_1 holds starting from the next moment after the *initial moment* of the time–domain of σ.

The definition 3.9 is rooted at its first clause,

$$\sigma \models \langle\!|E|\!\rangle \ \textbf{iff} \ [\![E]\!]\sigma(0) = \textbf{true} \quad ,$$

which says that a state–assertion E (denoted as $\langle\!|E|\!\rangle$ on the logic level) holds for a computation σ, iff it holds at the *initial moment* of the time–domain of σ.

3.4.2 Validity and satisfiability

We can now formalize the notion $L(\phi)$, which is the set of computations "accepted by" a formula ϕ. Given a set of variables V,

$$L(\phi) \equiv_{Def} \{\sigma \in Comp(V) \mid \sigma \models \phi\} \quad .$$

In general, $L(\phi) \subseteq Comp(V)$, i.e. a formula ϕ "accepts" some computation and "rejects" other ones.

There are cases of a formula ϕ, which accept any computation; in other words, they are valid independent of the interpretation of their variables. Such a formula is called a "valid formula" or a "tautology". Given a set of variables V, define

$$\phi \in LTL_{A\mathcal{L}} \text{ is a } \textit{tautology} \ \ \textbf{iff} \ L(\phi) = Comp(V) \quad ;$$

the fact that a formula ϕ is a tautology is denoted as $\models \phi$.

The opposite case is that a formula rejects all computations. A formula is called "satisfiable", if it accepts at least one computation. Given a set of variables V, define

$$\phi \in LTL_{A\mathcal{L}} \text{ is } \textit{satisfiable} \ \ \textbf{iff} \ L(\phi) \neq \emptyset \quad .$$

Validity and satisfiability are dual notions, as stated by the next lemma.

Lemma 3.5 (Validity and satisfiability) *Assume a set of variables V, formula $\phi \in LTL_{A\mathcal{L}}$. Then*

$$\begin{array}{rcll} L(\neg\phi) & = & Comp(V) - L(\phi) & (1) \\ \phi \textit{ is a tautology} & \textbf{iff} & \neg\phi \textit{ is not satisfiable} & (2) \end{array}$$

The first fact (1) of lemma 3.5 follows from the definition of $L(\phi)$; the second fact (2) is a special case of the first one.

The temporal operator **always** is also known under name "globally". A formula ϕ is called *globally valid* iff formula $\textbf{always}\,\phi$ is valid. The temporal logic language introduced in definition 3.8 contains only so–called "future time" operators, i.e. operators which do not refer to the past of the actual moment, but only to the future. For this type of temporal logic, we have the next lemma.

Lemma 3.6 (Global validity) *Assume a set of variables $\mathcal{V}$, formula $\phi \in LTL_{A\mathcal{L}}$. Then*

$$\models \phi \ \textbf{iff} \ \models \textbf{always}\,\phi \qquad (*),$$

i.e. a valid formula is globally valid, and (in particular) vice versa.

Proof of lemma 3.6 :

Proof of (*), $\Rightarrow$. Assume that (*),$\Rightarrow$ does not hold.

$\hookrightarrow$[def. of $\models$] (i) $\forall\sigma\,.\,\sigma \models \phi$ and (ii) $\exists\sigma_0\,.\,\sigma_0 \not\models \textbf{always}\,\phi$

$\hookrightarrow$[from (ii)] $\exists i_0\,.\,\sigma_0^{(i_0)} \not\models \phi$ (**)

$\hookrightarrow$[choose in (i) $\sigma_0^{(i_0)}$ for σ] $\sigma_0^{(i_0)} \models \phi$ (***)

$\hookrightarrow$[Contradiction between (**) and (***)] (*), $\Rightarrow$ is valid.

Proof of (*), $\Leftarrow$. Assume that (*), $\Leftarrow$ does not hold.

$\hookrightarrow$[def.of $\models$] (i) $\exists\sigma_0\,.\,\sigma_0 \not\models \phi$ and (ii) $\forall\sigma\,.\,\sigma \models \textbf{always}\,\phi$

$\hookrightarrow$[def. of **always**] $\forall i \geq 0\,\forall\sigma\,.\,\sigma^{(i)} \models \phi$

$\hookrightarrow$[in particular with $i = 0$ ($\sigma^{(0)} \equiv \sigma$)] $\forall\sigma\,.\,\sigma \models \phi$

$\hookrightarrow$[Contradiction to (i)] (*),$\Leftarrow$ is valid. (q.e.d.)

Following the terminology of [26], we define two formulas ϕ_1,$\phi_2 \in LTL_{A\mathcal{L}}$ to be *equivalent*, denoted as $\phi_1 \sim \phi_2$, by:

$$\phi_1 \sim \phi_2 \textbf{ iff } \models \phi_1 \leftrightarrow \phi_2 \quad ; \qquad \text{(equivalent formulas)}$$

we further define the formulas $\phi_1, \phi_2 \in LTL_{A\mathcal{L}}$ to be *congruent*, denoted as $\phi_1 \approx \phi_2$, by:

$$\phi_1 \approx \phi_2 \textbf{ iff } \models \textbf{always}\,(\phi_1 \leftrightarrow \phi_2) \quad . \qquad \text{(congruent formulas)}$$

From lemma 3.6 it follows immediately that for the temporal logic language defined in definition 3.8 the notions of equivalence and congruence coincide. It is easy to show that $\sim$ is an *equivalence relation* on the set of formulas (i.e., reflexive, symmetric, and transitive).

3.4.3 Formula schemes

Further insight into the mathematical properties of (linear) temporal logic can be gained by considering formula *schemes*. A formula scheme is built upon schematic variables taken from some domain $\mathcal{U}$ of schematic variables disjoint from the set of variables $\mathcal{V}$. These schematic variables represent yet unknown temporal logic sub–formulas. A formula scheme is constructed from these schematic variables by application of temporal operators [2].

Definition 3.10 (*LTL*–formula schemes) *The language LTL of temporal logic over some assertion language $A\mathcal{L}$ (introduced in definition 3.8) defines the class of LTL–formula schemes by the following grammar :*

$$\begin{array}{rcll} \Phi & \longrightarrow & u \mid (\Phi_1) & \textit{(0)} \\ & \mid & \ldots & \textit{(1)–(4)} \end{array}$$

where $u \in \mathcal{U}$ is a schematic variable, and the production sets (1)–(4) are defined as for LTL in definition 3.8, with Φ instead of ϕ.

Let *free*(Φ) denote the set of (free) schematic variables in the formula scheme Φ. For $u_1, \ldots, u_k \in$ *free*(Φ), $\phi_1, \ldots, \phi_k \in LTL_{A\mathcal{L}}$,

[2] Formula schemes will play a central role in chapter 6, where the so–called chaining rule 6.1 for LSTD is introduced.

$$\Phi[\phi_1/u_1, \ldots, \phi_k/u_k]$$

denotes the instantiated scheme, where the schematic variable $u_1, \ldots, u_k$ have been instantiated (substituted by) the corresponding *LTL*–formulas $\phi_1, \ldots, \phi_k$.

An instantiated scheme $\Phi[\phi_1/u_1, \ldots, \phi_k/u_k]$ is called *fully instantiated*, if $\{u_1, \ldots, u_k\} = \mathit{free}(\Phi)$.

The next lemma states that substitution of equivalent sub–formulas in a temporal logic formula yields an equivalent formula.

Lemma 3.7 (Substitution of equivalent sub–formulas) *Let ϕ_1, ϕ_2 be formulas in $LTL_{A\mathcal{L}}$ and Φ an LTL–scheme with one (free) schematic variable u. Then*

$$\phi_1 \sim \phi_2 \Rightarrow \Phi[\phi_1/u] \sim \Phi[\phi_2/u] \quad .$$

Note: In the special case of a single occurrence of the schematic variable u in the scheme Φ, the fully instantiated scheme $\Phi[\phi_1/u]$ can be regarded as a formula ϕ with sub–formula ϕ_1. A consequence of lemma 3.7 is that replacing sub–formula ϕ_1 by an equivalent sub–formula ϕ_2 yields an equivalent formula $\phi' \equiv_{Def} \Phi[\phi_2/u]$.

Lemma 3.7 can be used to establish the property of the temporal logic $LTL_{A\mathcal{L}}$, that it is possible to apply a sequence of equivalence–preserving transformations, by which the negation operator $\neg$ can be "pushed inwards". This is possible because of the following set of equivalences:

$$\begin{array}{ccc}
\neg(\phi_1 \wedge \phi_2) & \sim & \neg\phi_1 \vee \neg\phi_2 \\
\neg(\phi_1 \vee \phi_2) & \sim & \neg\phi_1 \wedge \neg\phi_2 \\
\neg\neg\phi_1 & \sim & \phi_1 \\
\neg\textbf{always}\ \phi_1 & \sim & \textbf{eventually}\ \neg\phi_1 \\
\neg\textbf{eventually}\ \phi_1 & \sim & \textbf{always}\ \neg\phi_1 \\
\neg(\phi_1\ \textbf{until}\ \phi_2) & \sim & \neg\phi_2\ \textbf{unless}\ (\neg\phi_1 \wedge \neg\phi_2) \\
\neg(\phi_1\ \textbf{unless}\ \phi_2) & \sim & \neg\phi_2\ \textbf{until}\ (\neg\phi_1 \wedge \neg\phi_2) \\
\neg\textbf{next}\ \phi_1 & \sim & \textbf{next}\ \neg\phi_1
\end{array}$$

An easy induction argument shows that each formula can be transformed in a finite numer of steps into an equivalent formula, where the negation operator occurs only immediately in front of an assertion formula.

Definition 3.11 (Negation normal form) *A formula $\phi \in LTL_{A\mathcal{L}}$ is said to be in negation normal form ($\neg$–NF), if for each subformula ϕ_1 of ϕ*

$$\phi_1 \equiv \neg\phi' \quad \Longrightarrow \quad \phi' \equiv \langle\!|E|\!\rangle, \text{ for some } E \in A\mathcal{L}.$$

Definition 3.12 (Negation free formula schemes) *A formula scheme Φ is called negation free, if it does not contain any negation .*

We will use in the following the notion that a formula ϕ_1 "is stronger than" or "implies" another formula ϕ_2.

Definition 3.13 (Formula implication) *Let $\phi_1, \phi_2 \in LTL_{A\mathcal{L}}$. Formula ϕ_1 is said to imply formula ϕ_2, denoted by*

$$\phi_1 \Rightarrow \phi_2 \quad ,$$

iff

$$\models \phi_1 \rightarrow \phi_2 \quad ,$$

i.e. iff $\phi_1 \rightarrow \phi_2$ is a tautology.

The temporal operators of *LTL* have – similar to the Boolean operators $\wedge, \vee$ – a *monotonic behavior*, which is stated in the next lemma [3].

Lemma 3.8 (Monotonicity of negation–free formulas) *Let $\phi_1, \phi_2 \in LTL_{A\mathcal{L}}$, Φ a negation free formula scheme with one schematic variable u. Then*

$$\textbf{always}\,(\phi_1 \rightarrow \phi_2) \Rightarrow \Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \quad .$$

Proof of lemma 3.8 . By definition of $\Rightarrow$(definition 3.13), we have to show that

$$\forall\sigma \ : \ \sigma \models \textbf{always}\,(\phi_1 \rightarrow \phi_2) \rightarrow \Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u]$$

We assume σ to be arbitrarily fixed, and establish that

[3] Similar results can be found in the standard literature on temporal logic; see in particular [24].

$$\sigma \models \textbf{always}\,(\phi_1 \rightarrow \phi_2) \qquad \text{(LHS)}$$

implies

$$\sigma \models \Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \qquad \text{(RHS)}$$

By definition, (LHS) means:

$$\forall i{\geq}0 \;:\; \sigma^{(i)} \models \phi_1 \rightarrow \sigma^{(i)} \models \phi_2 \qquad \text{(LHS')}$$

The proof is by induction on the height of the parse tree of Φ, which is denoted by $|\Phi|$.

- case $|\Phi| = 0$: Then, $\Phi \equiv u$, and

  $$\Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \equiv \phi_1 \rightarrow \phi_2 \quad ;$$

  hence RHS follows from LHS'.

- case $|\Phi| = 1$, $\Phi \equiv \phi' \wedge u$: Then,

  $$\Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \equiv \phi' \wedge \phi_1 \rightarrow \phi' \wedge \phi_2 \quad ;$$

  RHS follows from LHS' by the monotonicity of $\wedge$. Similar cases (e.g. $\Phi \equiv_{Def} u \wedge \phi'$) are omitted.

- case $|\Phi| = 1$, $\Phi \equiv \phi' \vee u$: Same argument as for case $\Phi \equiv_{Def} \phi' \wedge u$.

- case $|\Phi| = 1$, $\Phi \equiv \textbf{always}\,(u)$: Then,

  $$\Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \equiv \textbf{always}\,(\phi_1) \rightarrow \textbf{always}\,(\phi_2) \quad ;$$

  We have to show that

  $$\sigma \models \textbf{always}\,(\phi_1) \rightarrow \sigma \models \textbf{always}\,(\phi_2) \quad ;$$

  Assume that for some $\forall\; k{\geq}0$, $\sigma^{(k)} \models \phi_1$. From LHS follows (in particular) that $\forall\; k{\geq}0$, $\sigma^{(k)} \models \phi_2$, so $\sigma \models \textbf{always}\,(\phi_2)$.

- case $|\Phi| = 1$, $\Phi \equiv \textbf{eventually}\,(u)$: Then,

  $$\Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \equiv \textbf{eventually}\,(\phi_1) \rightarrow \textbf{eventually}\,(\phi_2) \quad .$$

We have to show that

$$\sigma \models \textbf{eventually}\,(\phi_1) \rightarrow \sigma \models \textbf{eventually}\,(\phi_2) \quad ;$$

Assume that for some $k{\geq}0$, $\sigma^{(k)} \models \phi_1$. From LHS follows (in particular) that $\sigma^{(k)} \models \phi_2$, so $\sigma \models \textbf{eventually}\,(\phi_2)$.

- case $|\Phi| = 1$, $\Phi \equiv \textbf{next}\,(u)$: Then,

$$\Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u] \equiv \textbf{next}\,(\phi_1) \rightarrow \textbf{next}\,(\phi_2) \quad .$$

We have to show that

$$\sigma \models \textbf{next}\,(\phi_1) \rightarrow \sigma \models \textbf{next}\,(\phi_2) \quad ;$$

Assume that for some $\sigma^{(1)} \models \phi_1$. From LHS follows (in particular) that $\sigma^{(1)} \models \phi_2$, so $\sigma \models \textbf{next}\,(\phi_2)$.

- case $|\Phi| = 1$, $\Phi \equiv \phi' \,\textbf{until}\, u$: We have to show that

$$\sigma \models \phi' \,\textbf{until}\, \phi_1 \rightarrow \sigma \models \phi' \,\textbf{until}\, \phi_2 \quad ;$$

Assume that for some $k{\geq}0$, $\sigma^{(k)} \models \phi_1$, and $\forall i, 0{\leq}i < k$, $\sigma^{(i)} \models \phi'$. From LHS follows (pointwise implication) that $\sigma^{(k)} \models \phi_2$, so $\sigma \models \phi' \,\textbf{until}\, \phi_2$.

- case $|\Phi| = 1$, $\Phi \equiv u \,\textbf{until}\, \phi'$: We have to show that

$$\sigma \models \phi_1 \,\textbf{until}\, \phi' \rightarrow \sigma \models \phi_2 \,\textbf{until}\, \phi' \quad ;$$

Assume that for some $k{\geq}0$, $\sigma^{(k)} \models \phi'$, and $\forall i, 0{\leq}i < k$, $\sigma^{(i)} \models \phi_1$. From LHS follows (in particular for the interval $0 \ldots k-1$) that $\forall i, 0{\leq}i < k$, $\sigma^{(i)} \models \phi_2$, so $\sigma \models \phi_2 \,\textbf{until}\, \phi'$.

- case $|\Phi| = 1$, $\Phi \equiv \phi' \,\textbf{unless}\, u$: We have to show that

$$\sigma \models \phi' \,\textbf{unless}\, \phi_1 \rightarrow \sigma \models \phi' \,\textbf{unless}\, \phi_2 \quad ;$$

Assume that for some $k{\geq}0$, $\sigma^{(k)} \models \phi_1$, and $\forall i, 0{\leq}i < k$, $\sigma^{(i)} \models \phi'$. Then, the argument of case $\Phi \equiv u \,\textbf{until}\, \phi'$ applies. Otherwise, $\sigma \models \textbf{always}\,(\phi')$, so RHS holds trivially.

- case $|\Phi| = 1$, $\Phi \equiv u \,\textbf{unless}\, \phi'$: We have to show that

$$\sigma \models \phi_1 \,\textbf{unless}\, \phi' \rightarrow \sigma \models \phi_2 \,\textbf{unless}\, \phi' \quad ;$$

Assume that for some $k{\geq}0$, $\sigma^{(k)} \models \phi'$, and $\forall i, 0{\leq}i < k$, $\sigma^{(i)} \models \phi_1$. Then, the argument of case $\Phi \equiv u \,\textbf{unless}\, \phi'$ applies. Otherwise, $\sigma \models \textbf{always}\,(u)$, then the argument of case $\Phi \equiv \textbf{always}\,(u)$applies.

Now, assume that the lemma 3.8 has been proven for all schemes up to a fixed depth $K \geq 1, K \equiv_{Def} |\Phi|$.

Then, one of the following cases applies:

- case $|\Phi| = K + 1$, $\Phi \equiv op_1(\Phi')$, where op_1 is a unary LTL operator. Then $|\Phi'| \leq K$, so by induction hypothesis LHS implies

$$\sigma \models \Phi'[\phi_1/u] \rightarrow \Phi'[\phi_2/u] \quad .$$

  Then by the same argument used in the proof of the induction base case,

$$\sigma \models op_1(\Phi'[\phi_1/u]) \rightarrow op_1(\Phi'[\phi_2/u]) \quad ,$$

  hence RHS follows.

- case $|\Phi| = K+1$, $\Phi \equiv \phi'\ op_2\ \Phi'$ or $\Phi \equiv \Phi'\ op_2\ \phi'$, where op_2 is a binary LTL operator. Then $|\Phi'| \leq K$, so by induction hypothesis LHS implies

$$\sigma \models \Phi'[\phi_1/u] \rightarrow \Phi'[\phi_2/u] \quad .$$

  Then by the same argument used in the proof of the induction base case, RHS follows.

This concludes the proof of lemma 3.8. (q.e.d.)

The next lemma is an immediate consequence of lemma 3.8.

Lemma 3.9 (Monotonicity of negation–free formulas) *Let $\phi_1, \phi_2 \in LTL_{A\mathcal{L}}$, Φ a negation free formula scheme with one schematic variable u.*

Then

$$\phi_1 \Rightarrow \phi_2$$

implies

$$\Phi[\phi_1/u] \Rightarrow \Phi[\phi_2/u] \quad .$$

Proof of lemma 3.9 . By lemma 3.6, we can conclude from the premise that

$$\forall \sigma \ : \ \sigma \models \textbf{always}\,(\phi_1 \rightarrow \phi_2) \qquad (*)$$

Assume that for some σ_0,

$$\neg(\sigma_0 \models \Phi[\phi_1/u] \rightarrow \Phi[\phi_2/u]) \quad . \qquad (\#)$$

From (*) follows in particular that

$$\sigma_0 \models \textbf{always}\,(\phi_1 \rightarrow \phi_2)$$

Since all premises are the same, (#) would contradict the statement of lemma 3.8; hence the conclusion of lemma 3.9 follows.

3.5 Sub–logics of Temporal Logic

The formalisms introduced so far (LTL, POSA) are sufficient to explain the semantics of STD.

However, in order to prepare a main theorem about STD (linear decomposition), we need to go a step further, anticipating the definition of the structure of STD–specifications.

STD–specifications are *sets* of diagrams, with a conjunctive interpretation of the set–constructor. The semantics of a STD–specification *SPEC* has therefore the following structure in LTL:

$$\phi_{SPEC} = \phi_1^\alpha \wedge \ldots \wedge \phi_k^\alpha$$

where ϕ_i^αdescribes the semantics of a particular diagram in the set.

A STD–diagram has one of two possible interpretations (called *activation modes*):

- initial, in which case the semantics of the diagram body is bound to a particular moment in time, namely the very first step (step 0)
- invariant, in which case the semantics of the diagram is evaluated in *every step* of a computation.

It follows that the semantics of an STD–diagram has either the form ϕ or **always** (ϕ), where ϕ is a characterization of the diagram body.

The semantics of an STD–diagram body can be characterized by a nested **until** / **unless** –formula; more precisely, nesting occurs only in the second argument of the **until** / **unless** –formula, yielding the form:

$$\phi_1 \,\mathcal{U}\, (\ldots \phi_2 \,\mathcal{U}\, (\ldots \phi_3 \,\mathcal{U}\, \ldots) \ldots) \quad (*)$$

where $\mathcal{U}$ is either **until** or **unless**.

For an even more detailed discussion, we need to distinguish between purely Boolean formulas – henceforth denoted by letter $\beta \in Bool_{A\mathcal{L}}$ – and "really" temporal logic formulas – denoted by letter ϕ.

Given the distinction between Boolean– and temporal–logic formulas, we will show later in this section (in full detail in chapter 6) that the right argument of the **until** –respectively **unless** –operator in the formula $(*)$ has the form

$$\bigvee_{i=1\ldots k} (\beta_i \wedge \phi_i)$$

i.e. it is a finite disjunction of conjuncts ϕ_i with "guard" β_i.

Moreover, in the case of the STD–semantics, the guards β_i are pairwise disjoint (i.e., mutually exclusive); the same restriction applies between the first and the second argument of each $\mathcal{U}$ –operator.

We will call this logic *restricted* LTL (over set of variables V), denoted LTL_V^-.

Definition 3.14 (Restricted (linear–time) temporal logic) *Assume an assertion language $A\mathcal{L}$, and a set V of variables. Then the language of restricted linear–time temporal logic based on $A\mathcal{L}(LTL_V^-)$ is defined by the following grammar with the production sets (1)–(7):*

$$\begin{array}{rcll}
\phi^\alpha & \longrightarrow & \phi_1^\alpha \wedge \phi_2^\alpha & (1)\\
 & | & \phi_1 \;|\, \mathbf{always}\, \phi_1 & (2)\\
\phi & \longrightarrow & \beta_1 \;|\, (\phi_1) & (3)\\
 & | & \bigvee_{i=1\ldots k} (\beta_i \wedge \phi_i) \quad (k>0) & (4)\\
 & & \textit{where all } \beta_i \textit{ are pairwise disjoint} & \\
 & & (\beta_i \Rightarrow \neg\beta_j, i \neq j) & \\
 & | & \beta_1 \,\mathbf{until}\, (\neg\beta_1 \wedge \phi_1) & (5)\\
 & | & \beta_1 \,\mathbf{unless}\, (\neg\beta_1 \wedge \phi_1) & (6)\\
\beta & \longrightarrow & \neg\beta_1 \;|\, \beta_1 \wedge \beta_2 \;|\, \beta_1 \vee \beta_2 \;|\, \langle\!|E|\!\rangle & (7)
\end{array}$$

where E is a Boolean expression of the assertion language $A\mathcal{L}$ over V.

Remark. *For readers familiar with the notions of linear–time and branching–time temporal logic, we note the following fact: The linear–time and the branching–time semantics are equivalent for formulas of LTL_V^-, if in case of the branching time semantics (where formulas are interpreted relative to computation trees instead of computations) the temporal operators are interpreted as follows: "For all paths starting from the root of the computation tree, ...". This means in particular that the efficient branching–time model–checking procedure (with a linear complexity in the size of the formula) can be used for model verification with STD, although STD has a linear–time semantics definition. The interested reader is referred to [4] for full details on the comparison between the semantics of linear–time and branching–time temporal logic.*

It turns out that formulas in LTL_V^- can be syntactically rewritten to a normal form, which will be exploited later to obtain a normal form for the graphical presentation of STD.

The next definition avoids disjunction of temporal formulas.

Definition 3.15 (Restricted (linear–time) temporal logic without temporal disjunction) *Assume an assertion language $A\mathcal{L}$, and a set V of variables. Then the language of restricted linear–time temporal logic without temporal disjunction based on $A\mathcal{L}(LTL_V^{-\wedge})$ is defined by the following grammar with the production sets (1)–(8):*

$$
\begin{array}{lcll}
\phi^\alpha & \longrightarrow & \phi_1^\alpha \wedge \phi_2^\alpha & (1)\\
 & | & \phi_1 \mid \textbf{always}\, \phi_1 & (2)\\
\phi & \longrightarrow & ((\beta_1 \wedge \phi_1) \vee \beta_2) & (3)\\
 & | & \beta_1\ \textbf{until}\quad ((\beta_2 \wedge \phi_1) \vee \beta_3) & (4)\\
 & | & \beta_1\ \textbf{unless}\quad ((\beta_2 \wedge \phi_1) \vee \beta_3) & (5)\\
 & | & \beta_1 \mid (\phi_1) & (6)\\
 & | & \phi_1 \wedge \phi_2 & (7)\\
\beta & \longrightarrow & \neg\beta_1 \mid \beta_1 \wedge \beta_2 \mid \beta_1 \vee \beta_2 \mid \langle\!|E|\!\rangle & (8)
\end{array}
$$

where E is a Boolean expression of the assertion language $A\mathcal{L}$ over $\mathcal{V}$, and $\beta_1, \beta_2, \beta_3$ are disjoint in production (3)–(5).

The next lemma is the key to the first normalization transformation.

Lemma 3.10 (Transformation of temporal disjunction to conjunction) *Assume an assertion language $A\mathcal{L}$, and a set V of variables.*

Assume Boolean expressions $\beta_1 \ldots, \beta_k$ (for some $k > 0$), and LTL_V^- formulas $\phi_1, \ldots, \phi_k$ over V.

Define formulas ϕ_A and ϕ_B as follows:

$$\phi_A \equiv_{Def} \bigvee_{j=1\ldots k} (\beta_j \wedge \phi_j)$$

and

$$\phi_B \equiv_{Def} ((\beta_1 \wedge \phi_1) \vee \bigvee_{j=1\ldots k, j\neq 1} \beta_j) \quad \wedge \quad \ldots \quad \wedge \quad ((\beta_k \wedge \phi_k) \vee \bigvee_{j=1\ldots k, j\neq k} \beta_j)$$

If the Boolean expressions $\beta_1 \ldots, \beta_k$ are pairwise disjoint (cf. def. 3.14), then formulas ϕ_A and ϕ_B are equivalent, i.e. $\phi_A \sim \phi_B$.

Proof of lemma 3.10 . We assume an arbitrary fixed computation sequence σ.

- $\phi_A \Rightarrow \phi_B$: Assume that $\sigma \models \phi_A$; show that $\sigma \models \phi_B$. $\sigma \models \phi_A$ implies $\sigma \models (\beta_{i_0} \wedge \phi_{i_0})$, for some i_0. Then $\sigma \models (\beta_{i_0} \wedge \phi_{i_0})$ in the conjunct with index i_0 of ϕ_B. The other conjuncts with index $r \neq i_0$ are also satisfied, because $\bigvee_{j=1\ldots k, j\neq r} \beta_j$ contains β_{i_0}.

- $\phi_B \Rightarrow \phi_A$: Assume by contrary that $\sigma \models \phi_B$ and $\sigma \not\models \phi_A$. $\sigma \not\models \phi_A$ means that $\forall i \ : \ \sigma \not\models (\beta_i \wedge \phi_i)$. Since $\sigma \models \phi_B$, this means that $\forall i \ : \ \sigma \models \bigvee_{j=1\ldots k, j\neq i} \beta_j$. It is easy to see that this is impossible under the assumption that the Boolean expressions $\beta_1 \ldots, \beta_k$ are pairwise disjoint, hence the implication $\phi_B \Rightarrow \phi_A$ must hold.

q.e.d.

The next theorem shows that each formula of LTL_V^- can be transformed into an equivalent formula in $LTL_V^{-\wedge}$.

Theorem 3.1 (Transformation of LTL_V^- in $LTL_V^{-\wedge}$) *For each formula $\phi \in LTL_V^-$, there is an equivalent formula $\phi' \in LTL_V^{-\wedge}$.*

Proof of theorem 3.1 . The idea of the proof is that for a production sequence in LTL_V^- which leads to formula ϕ, there is a corresponding – semantics preserving – production sequence in $LTL_V^{-\wedge}$, which leads to formula ϕ'. *Semantics preserving* means: Each derivation step in the generation of formula ϕ consists of replacing a subformula $\tilde{\phi}$ of ϕ by the right side of the selected production. The right side may include 0 or more temporal formulas, i.e. it is a term $\Phi(\phi_1, \ldots, \phi_k)$, for some $k \geq 0$.

For each such production step, we can find a finite production (sub–)sequence in $LTL_V^{-\wedge}$, which transforms $\tilde{\phi}$ into an equivalent term $\Phi'(\phi'_1, \ldots, \phi'_k)$ in $LTL_V^{-\wedge}$. with subformulas $\phi'_1, \ldots, \phi'_k$.

For each of the subformulas ϕ_i, similar transformations are applied, yielding eventually a finite production sequence in $LTL_V^{-\wedge}$ of a formula ϕ'_i, which is equivalent to ϕ_i.

- case production sets (1),(2) in LTL_V^-: There exist corresponding productions in $LTL_V^{-\wedge}$.
- case production set (3) in LTL_V^-: There exist corresponding productions (set 6) in $LTL_V^{-\wedge}$.
- case production set (4) in LTL_V^-: Assume that we expand subformula $\tilde{\phi}$ using the production:

  $$\tilde{\phi} \longrightarrow \bigvee_{i=1\ldots k} (\beta_i \wedge \phi_1)$$

  By lemma 3.10, the production

  $$\tilde{\phi} \longrightarrow \bigwedge_{i=1\ldots k} ((\beta_i \wedge \phi_i) \vee \bigvee_{j=1\ldots k, j\neq i} \beta_j)$$

  produces an equivalent subformula, which can be generated in $LTL_V^{-\wedge}$ by $(k-1)$ applications of production (7), k applications of production (3), and the generation of the Boolean formula $\bigvee \beta_j$.
- case production set (5) in LTL_V^-: Assume that we expand subformula $\tilde{\phi}$ using the production:

  $$\tilde{\phi} \longrightarrow \beta_1 \textbf{ until } (\neg\beta_1 \wedge \phi_1)$$

  With $\beta_2 \equiv_{Def} \neg\beta_1$, $\beta_3 \equiv_{Def} \langle\!|False|\!\rangle$ we can use production (4) in $LTL_V^{-\wedge}$ to obtain an equivalent subformula. Note, that $\beta_1, \beta_2, \beta_3$ with the above definitions are pairwise disjoint.

- case production set (6) in LTL_V^-: This case is handled similarly, using production (5) in $LTL_V^{-\wedge}$ to obtain an equivalent subformula.
- case production set (7) in LTL_V^-: There exist corresponding productions (set 8) in $LTL_V^{-\wedge}$.

q.e.d.

The next lemma is the key to another normalization step, "lifting" conjunction of temporal formulas to the top–level of the specification formula.

Lemma 3.11 (Outwards conjunction transformation) *Assume an assertion language $A\mathcal{L}$, and a set V of variables.*

Assume Boolean expression β and LTL_V formulas ϕ_1, ϕ_2 over V.

Define formulas ϕ_A and ϕ_B as follows:

$$\phi_A \equiv_{Def} \quad \beta \,\mathcal{U}\, (\neg\beta \wedge \phi_1 \wedge \phi_2)$$

and

$$\phi_B \equiv_{Def} \quad \beta \,\mathcal{U}_1 (\neg\beta \wedge \phi_1) \wedge \beta \,\mathcal{U}_2 (\neg\beta \wedge \phi_2)$$

Then the following facts hold:

1. *case* $\mathcal{U} = \mathcal{U}_1 = \mathcal{U}_2 =$ **unless** *:* $\phi_A \sim \phi_B$.
2. *case* $\mathcal{U} = \mathcal{U}_1 = \mathcal{U}_2 =$ **until** *:* $\phi_A \sim \phi_B$.
3. *case* $\mathcal{U} = \mathcal{U}_1 =$ **until** , $\mathcal{U}_2 =$ **unless** *:* $\phi_A \sim \phi_B$.

Proof of lemma 3.11 . The implication $\phi_A \Rightarrow \phi_B$ follows in all three cases by expansion of the definition of the $\mathcal{U}$ operator.

We consider the interesting case 3, $\phi_B \Rightarrow \phi_A$; the implication is proven similarly for cases 1 and 2.

Assume an arbitrary fixed computation sequence σ.

$\sigma \models \phi_B$

$\hookrightarrow$[Def. of ϕ_B] $\sigma \models \beta$ **until** $(\neg\beta \wedge \phi_1) \wedge \sigma \models \beta$ **unless** $(\neg\beta \wedge \phi_2)$

$\hookrightarrow$[Def. **until** , **unless**]

$\exists l_1 \geq 0 \,.\, \sigma^{(l_1)} \models (\neg\beta \wedge \phi_1)$ and $\exists l_2 \geq 0 \,.\, \sigma^{(l_2)} \models (\neg\beta \wedge \phi_2)$
and $\forall i \,.\, 0 \leq i < max\{l_1, l_2\} \;:\; \sigma(i) \models \beta$.

This is only possible if $l_1 = l_2 \equiv_{Def} l$, because $\sigma(l_i) \models \neg\beta$(i=1,2). Hence, $\sigma^{(l)} \models (\neg\beta \wedge \phi_1 \wedge \phi_2)$, and therefore $\sigma \models \phi_A$. q.e.d.

Our goal is the transformation of LTL_V formulas into a logic called $LINLTL_V^-$, which is defined next (definition 3.16).

The definition of $LINLTL_V$ reintroduces the **next** –operator which will be needed in chapter 5. Here we are interested in the **next** –free fragment of the definition, called $LINLTL_V^-$. As in the case of LTL_V^-, $LINLTL_V^-$ avoids the nondeterminism which is inherent in the semantics definition of the temporal **until** and **unless** operator.

Definition 3.16 ((Restricted) sequencing (linear–time) temporal logic) *Assume an assertion language $A\mathcal{L}$, and a set $\mathcal{V}$ of variables. Then the language of sequencing linear–time temporal logic based on $A\mathcal{L}(LINLTL_V)$ is defined by the following grammar with the production sets (1)–(8):*

$$
\begin{array}{lcll}
\xi^\alpha & \longrightarrow & \xi_1^\alpha \wedge \xi_2^\alpha & (1)\\
 & | & \xi_1 \;|\, \textbf{always}\, \xi_1 & (2)\\
\xi & \longrightarrow & ((\beta_1 \wedge \psi_1) \vee \beta_2) & (3)\\
 & | & \beta_1 \;\textbf{until}\; ((\beta_2 \wedge \psi_1) \vee \beta_3) & (4)\\
 & | & \beta_1 \;\textbf{unless}\; ((\beta_2 \wedge \psi_1) \vee \beta_3) & (5)\\
 & | & \beta_1 & (6)\\
\psi & \longrightarrow & \xi \;|\, \textbf{next}\, \xi & (7)\\
\beta & \longrightarrow & \neg\beta_1 \;\; |\, \beta_1 \wedge \beta_2 \;\; |\, \beta_1 \vee \beta_2 \;\; |\, \langle\!|E|\!\rangle & (8)
\end{array}
$$

where E is a Boolean expression of the assertion language $A\mathcal{L}$ over $\mathcal{V}$.

The logic of restricted sequencing linear–time temporal logic based on $A\mathcal{L}(LINLTL_V^-)$ is defined by the same grammar, except for the following restrictions:

1. *In clause (3)–(5), all Boolean formulas β_i are disjoint, and*
2. *in clause (7), only the production $\psi \longrightarrow \xi$ is allowed, i.e., the logic $LINLTL_V^-$ has no* **next** *–operator.*

Theorem 3.2 (Transformation of $LTL_V^{-\wedge}$ in $LINLTL_V^-$) *For each formula $\phi \in LTL_V^{-\wedge}$, there is an equivalent formula $\phi' \in LINLTL_V^-$.*

Proof of theorem 3.2 . The idea of the proof is similar to the argument used to show the existence of the negation normal form for LTL–formulas by a series of equivalences concerning the negation operator in front of temporal operators (cf. def.3.11).

For each production result in the definition of $LTL_V^{-\wedge}$, we show that an occurrence of the conjunction operator in the production term can be moved (one level) "outwards".

- case **always** (ϕ): **always** $(\phi_1 \wedge \phi_2) \sim$ **always** $(\phi_1) \wedge$ **always** (ϕ_2)
- case $(\beta_1 \wedge \phi) \vee \beta_2$: $(\beta_1 \wedge (\phi_1 \wedge \phi_2)) \vee \beta_2 \sim ((\beta_1 \wedge \phi_1) \vee \beta_2) \wedge ((\beta_1 \wedge \phi_2) \vee \beta_2)$
- case $\beta_1 \mathcal{U} ((\beta_2 \wedge \phi) \vee \beta_3)$, $\mathcal{U} \in \{$ **until** , **unless** $\}$:

$$
\begin{array}{rc}
 & \beta_1 \mathcal{U} ((\beta_2 \wedge (\phi_1 \wedge \phi_2)) \vee \beta_3) \\
\sim & [\beta_i \Rightarrow \neg\beta_1, i = 2,3] \\
 & \beta_1 \mathcal{U} (\neg\beta_1 \wedge ((\beta_2 \wedge (\phi_1 \wedge \phi_2)) \vee \beta_3)) \\
\sim & \beta_1 \mathcal{U} (\neg\beta_1 \wedge ((\beta_2 \wedge \phi_1) \vee \beta_3) \wedge ((\beta_2 \wedge \phi_2) \vee \beta_3)) \\
\sim & [\text{lemma } 3.11] \\
 & \beta_1 \mathcal{U} (\neg\beta_1 \wedge ((\beta_2 \wedge \phi_1) \vee \beta_3)) \wedge \beta_1 \mathcal{U} (\neg\beta_1 \wedge ((\beta_2 \wedge \phi_2) \vee \beta_3)) \\
\sim & [\beta_i \Rightarrow \neg\beta_1, i = 2,3] \\
 & \beta_1 \mathcal{U} ((\beta_2 \wedge \phi_1) \vee \beta_3) \wedge \beta_1 \mathcal{U} ((\beta_2 \wedge \phi_2) \vee \beta_3) \quad .
\end{array}
$$

q.e.d. From theorem 3.1 and 3.2 we get the next important result.

Corollary 3.1 (Transformation of LTL_V^- in $LINLTL_V^-$) *For each formula $\phi \in LTL_V^-$, there is an equivalent formula $\phi' \in LINLTL_V^-$.*

Since $LINLTL_V^-$ is a syntactical subset of LTL_V^-, is follows that both logics have the same expressive power.

Note that the introduction of the logic–formalisms $LTL_V^{-\wedge}$ and $LINLTL_V^-$ has the only purpose to provide the basis for the decomposition theorem 6.2 from STD to LSTD introduced in chapter 6. The logics have not been further analyzed in the context of this thesis with respect to expressiveness and their ability to be used as specification formalisms [4].

[4] See appendix B for further notes on this topic.

3.6 Translation from Symbolic Automata to Temporal Logic

So far, we have introduced two quite different formalisms for specification: Symbolic automata, which are *operational* in nature, and temporal logic, which is *declarative* in nature.

3.6.1 Temporal logic characterization of POSA

The central observation made in this section is that there is a close connection between the sub–class of partially–ordered symbolic automata and temporal logic.

In particular it is shown that the semantics of a POSA can be characterized in terms of linear temporal logic.

Theorem 3.3 (Temporal logic characterization of POSA) *Assume an assertion language $A\mathcal{L}$, and let $\mathcal{A}$ be a POSA over some set V of variables. Then there exists a formula $\phi_{\mathcal{A}} \in LTL_{A\mathcal{L}}$ over V such that*

$$L(\mathcal{A}) = L(\phi_{\mathcal{A}}) \quad . \qquad (*)$$

Construction. According to lemma 3.4 , we can assume that $\mathcal{A}$ is normalized.

Let $\mathcal{A} \equiv_{Def} (V, Locs, Edges, L_0, F)$. We construct $\phi_{\mathcal{A}}$ as follows: For each pair of locations $\ell, \ell' \in Locs$, $\ell \to \ell'$, let

$$\phi_{\ell,\ell'} =_{Def} \textbf{any } \phi \,.\, (\ell, \phi, \ell') \in Edges \quad .$$

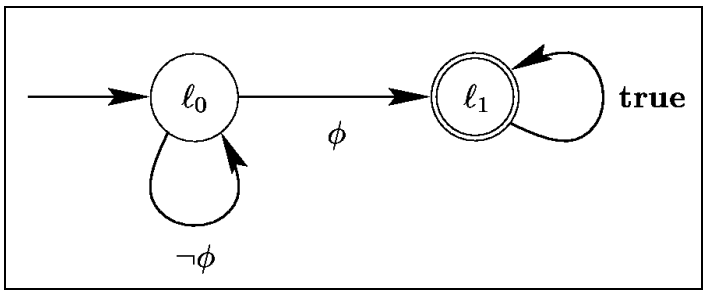
Note that $\mathcal{A}$ is normalized, so in particular (1) $\ell \to \ell$ (existence of self–loops) and (2) no parallel edges exist; hence $|\{\phi \mid (\ell, \phi, \ell') \in Edges\}| = 1$, which means that $\phi_{\ell,\ell'}$ is uniquely defined for all $\ell, \ell' \in Locs$.

Define for all locations $\ell \in Locs$

$$\phi\chi_\ell =_{Def} \phi_{\ell,\ell} \quad \mathcal{U} \quad (\bigvee_{\ell \neq \ell' \,:\, \ell \to \ell'} \phi_{\ell,\ell'} \wedge \textbf{next}\, \phi_{\ell'})$$

where

Figure 3.2: Symbolic Automaton $\mathcal{A}_{\textbf{eventually } \phi}$.



$$\begin{array}{lll} \mathcal{U} & =_{Def} \quad \textbf{unless} & \text{if } \ell \in F, \text{ and} \\ \mathcal{U} & =_{Def} \quad \textbf{until} & \text{if } \ell \notin F \end{array}$$

and let

$$\phi_{\mathcal{A}} =_{Def} \bigvee_{\ell \in L_0} \phi \chi_\ell \quad .$$

(Note that $\mathcal{A}$ is assumed to be normalized, hence complete; therefore the disjunction in the definitions of ϕ_ℓ and $\phi_{\mathcal{A}}$ is always over nonempty sets.) Given these definitions, we claim that (*) holds.

Example 3.3 (Translation of SA to temporal logic) *We reconsider the automaton*

$$\mathcal{A}_{\textbf{eventually } \phi} = (V, Locs, Edges, L_0, F)$$

where the components Locs, Edges, L_0, F are specified in diagrammatic notation in figure 3.2.

Note that $\ell_1 \in F$ is an accepting state, whereas ℓ_0 is not an accepting state.

According to theorem 3.3 we first construct the basic formula according to transition labels:

$$\begin{array}{rcl} \phi_{\ell_0,\ell_0} & = & \neg\phi \\ \phi_{\ell_0,\ell_1} & = & \phi \\ \phi_{\ell_1,\ell_1} & = & \mathbf{true} \end{array}$$

Next, we construct the formulas characterizing the states:

$$\begin{array}{rcl} \phi_{\ell_0} & = & \phi_{\ell_0,\ell_0}\ \mathbf{until}\ (\phi_{\ell_0,\ell_1} \wedge \mathbf{next}\ \phi_{\ell_1}) \\ \phi_{\ell_1} & = & \phi_{\ell_1,\ell_1}\ \mathbf{unless\ false} \end{array}$$

Putting all the parts together, we obtain

$$\begin{array}{rcll} \phi_{\mathcal{A}} & = & & \phi_{\ell_0} \\ & = & & \phi_{\ell_0,\ell_0}\ \mathbf{until}\ (\phi_{\ell_0,\ell_1} \wedge \mathbf{next}\ \phi_{\ell_1}) \\ & = & & \phi_{\ell_0,\ell_0}\ \mathbf{until}\ (\phi_{\ell_0,\ell_1} \wedge \mathbf{next}\ (\phi_{\ell_1,\ell_1}\ \mathbf{unless\ false})) \\ & = & & \neg\phi\ \mathbf{until}\ (\phi \wedge \mathbf{next}\ (\mathbf{true\ unless\ false})) \\ & = & \textit{[(1,2)]} & \neg\phi\ \mathbf{until}\ \phi \\ & = & \textit{[(3)]} & \mathbf{eventually}\ \phi \end{array}$$

where the following equivalences where used to simplify the formulas

$$\begin{array}{rcll} (\mathbf{true\ unless}\ \phi) & \sim & \mathbf{true} & \textit{(1)} \\ \mathbf{next\ true} & \sim & \mathbf{true} & \textit{(2)} \\ \neg\phi\ \mathbf{until}\ \phi & \sim & \mathbf{eventually}\ \phi & \textit{(3)} \end{array}$$

Proof of theorem 3.3 . It suffices to consider those locations $\ell \in \mathit{Locs}$, which can be reached by a run starting from some initial location $\ell_0 \in L_0$; we denote these "reachable" locations by the set

$$R \equiv_{Def} R_{L_0,\preceq} =_{Def} \{\ell \in \mathit{Locs} \mid \exists \ell_0 \in L_0 \,.\, \ell_0 \preceq \ell\} \quad ;$$

obviously (by definition of R, reflexivity of $\preceq$), $L_0 \subseteq R$.

The main idea of the proof is to define a monotone sequence $(T_i)_{i\geq 0}$ of sets such that

$$T_i =_{Def} \{\ell \in R \mid \Delta T(\ell) \leq i\} \qquad (T_i\text{–def})$$

where

$$\begin{array}{rcl}
T \equiv_{Def} T_{L_0,\preceq} & =_{Def} & \{\ell \in R \mid \neg\exists \ell' \neq \ell \,.\, \ell \rightarrow \ell'\} \\
\delta(\ell, \ell_f) & =_{Def} & \mathbf{max}\,\{|D| - 1 \mid D \subseteq R, \text{ and } D \text{ is an} \\
 & & \quad \rightarrow\text{–chain with first element } \ell \text{ and} \\
 & & \quad \text{last element } \ell_f\} \quad \text{if } \ell \preceq \ell_f, \\
\delta(\ell, \ell_f) & =_{Def} & \quad \bot \text{ (undefined)} \quad \text{else ;} \\
\Delta T(\ell) & =_{Def} & \mathbf{max}\,\{\delta(\ell, \ell_f) \mid \ell_f \in T, \ell \preceq \ell_f\}
\end{array}$$

The set T contains the maximal ("terminal") elements of the set of reachable locations R with respect to the partial order $\preceq$. For a location $\ell \in R$, $\Delta T(\ell)$ defines the "maximal number of remaining $\rightarrow$–steps" between $\ell \in R$ and the terminal element set T.

The next lemma analyzes the essential properties of (T_i) implied by this definition.

Lemma 3.12 (Properties of (T_i)) *Under the assumptions of theorem 3.3, let the sequence $(T_i)_{i\geq 0}$ be defined by definition (T_i–def). Then*

$$\begin{array}{ll}
\textit{(1)} & T = \{\ell \in R \mid \ell \textit{ is a maximal element w.r.t. } \preceq_{|R}\} \\
\textit{(2)} & \forall \ell \in R \,:\, \Delta T(\ell) \geq 0 \\
\textit{(3)} & T = T_0 \\
\textit{(4)} & \exists i \geq 0 \,.\, T_0 \subseteq T_1 \subseteq \ldots \subseteq T_i = R \textit{ and } \forall k \geq 0 \,:\, T_i = T_{i+k} \\
\textit{(5)} & \forall i \geq 0 \forall \ell, \ell' \in \mathit{Locs} \,.\, \ell \neq \ell' \,:\, \ell \in T_{i+1} \wedge \ell \rightarrow \ell' \implies \ell' \in T_i
\end{array}$$

where the property (5) is the key for the inductive argument used in the proof of theorem 3.3.

Proof of lemma 3.12 .

Proof of (1) $T \subseteq RHS(1)$ **.** Let $\ell \in T$ $(*)$ and assume that $\ell \notin RHS(1)$

$\hookrightarrow$[ℓ is not maximal w.r.t. $\preceq$: $\exists \ell\text{“} \neq \ell \,.\, \ell \preceq \ell\text{“}$

$\implies \quad \exists \ell' \neq \ell \,.\, \ell \rightarrow \ell']$ $\ell \notin T$

$\hookrightarrow$[contradiction to $(*)$] $\ell \in RHS(1)$.

$RHS(1) \backslash T = \emptyset$. Let $\ell \in R \backslash T$;
$\hookrightarrow$[$\exists \ell' \neq \ell \,.\, \ell \rightarrow \ell'$, so $\ell \preceq \ell'$] $\ell \notin RHS(1)$.

Proof of (2). Let $\ell \in R$ be arbitrarily fixed;
$\hookrightarrow$[by definition of the set R] $\exists \ell_0 \in L_0 \,.\, \ell_0 \preceq \ell$.

$\hookrightarrow$[def. of $\preceq$] $\exists k \geq 0, \exists \{\ell_0, \ldots, \ell_k\} \subseteq Locs \,.\, \ell_0 \rightarrow \ldots \rightarrow \ell_k = \ell$

$\hookrightarrow$[with $D \equiv_{Def} \{\ell_0, \ldots, \ell_k\}$, D is a $\rightarrow$–chain
whose first element is a minimal element of $\preceq$]
$\exists d \geq 0, \bar{D} \equiv_{Def} \{\ell_0, \ldots, \ell_{k+d}\}$.
(1) $D \subseteq \bar{D} \subseteq R$,
(2) $\bar{D}$ is a $\rightarrow$–chain, and
(3) the last element ℓ_{k+d} of $\bar{D}$ is a maximal element of $\preceq$

$\hookrightarrow$[prop.(1)]
$\exists \ell_0 \in L_0, \ell_{max} \in T \,.\, \ell_0 \rightarrow \ldots \rightarrow \ell_k = \ell \rightarrow \ldots \rightarrow \ell_{k+d} = \ell_{max}$.

$\hookrightarrow$[def. of $\delta(\ell, \ell_{max})$, $\ell \preceq \ell_{max}$, $\ell_{max} \in T$] $\Delta T(\ell) \geq \delta(\ell, \ell_{max}) \geq d \geq 0$.

Proof of (3). Show that: $T = T_0$, where $T_0 = \{\ell \in R \mid \Delta T(\ell) \leq 0\}$.
$T \subseteq T_0$: Let $\ell \in T$; show that $\Delta T(\ell) = 0$.

$\hookrightarrow$[$\ell \in T$] $\ell \in R$ **and** $\neg \exists \ell' \neq \ell \,.\, \ell \rightarrow \ell'$ $(*)$

Assume that $\Delta T(\ell) > 0$

$\hookrightarrow$[def. of $\Delta T(\ell)$] $\exists \ell_f \in T \,.\, \ell \preceq \ell_f \wedge \delta(\ell, \ell_f) > 0$

$\hookrightarrow$[def. of $\delta(\ell, \ell_f)$] $\exists k > 0, D \equiv_{Def} \{\ell_0, \ldots, \ell_k\}$.
(1) $D \subseteq R$,
(2) D is a $\rightarrow$–chain s.t. $\ell = \ell_0 \rightarrow \ell_1 \rightarrow \ldots \rightarrow \ell_k = \ell_f$

$\hookrightarrow$[$k > 0$] $\exists \ell' = \ell_1 \neq \ell \,.\, \ell \rightarrow \ell'$

which contradicts $(*)$; hence it must hold that $\neg(\Delta T(\ell) > 0)$.

$\hookrightarrow$[property (2): $\Delta T(\ell) \geq 0$] $\Delta T(\ell) = 0$.

$T_0 \subseteq T$: Let $\ell \in T_0$, i.e. $\Delta T(\ell) \leq 0$.

$\hookrightarrow$[property (2): $\Delta T(\ell) \geq 0$] $\Delta T(\ell) = 0$ (#)

Assume that ℓ were not a maximal element w.r.t. $\preceq_{|R}$

$\hookrightarrow$[ℓ is not maximal w.r.t. $\preceq$: $\exists \ell\text{“} \neq \ell \,.\, \ell \preceq \ell\text{“}$

$\quad \Longrightarrow \quad \exists \ell' \neq \ell \,.\, \ell \rightarrow \ell']$ $\Delta T(\ell) \geq 1$

$\hookrightarrow$[contradiction to (#)] ℓ is a maximal element w.r.t. $\preceq_{|R}$

$\hookrightarrow$[property (1)] $\ell \in T$.

Proof of (4). From the definition $T_i =_{Def} \{\ell \in R \mid \Delta T(\ell) \leq i\}$ follows immediately the monotonicity of the sequence (T_i) within the set R, i.e.

$$\forall i \geq 0 \,:\, T_i \subseteq T_{i+1} \wedge T_i \subseteq R \quad .$$

Let $i_R =_{Def} \mathbf{max}\,\{\Delta T(\ell) \mid \ell \in R\}$; show that

$\hookrightarrow$[$\forall \ell \in R \,:\, \Delta T(\ell) \leq i_R$] $\forall k \geq 0 \,:\, R \subseteq T_{i_R+k}$

$\hookrightarrow$[$T_{i+k} \subseteq R \subseteq T_{i+k}$, monotonicity of (T_i)]

$\quad T_0 \subseteq T_1 \subseteq \ldots \subseteq T_{i_R} = R$ **and** $\forall k \geq 0 \,:\, R = T_{i_R+k}$

$\hookrightarrow$[$\exists i \equiv_{Def} i_R$] (4).

Proof of (5). Assume arbitrary fixed number $i \geq 0$, location $\ell, \ell' \in Locs$ such that $\ell \neq \ell'$; show that

$$\ell \in T_{i+1} \wedge \ell \rightarrow \ell' \quad \Longrightarrow \quad \ell' \in T_i$$

Let $\ell \in T_{i+1}$

$\hookrightarrow$[def. of T_{i+1}] $\Delta T(\ell) \leq i + 1$

Assume further that $\ell \rightarrow \ell'$; we want to show that $\Delta T(\ell') \leq i$ follows.
Assume by contrary that $\Delta T(\ell') > i$

$\hookrightarrow$[def. of $\Delta T(\ell')$] $\exists k > i \; \exists \ell_f \in T \,.\, \delta(\ell', \ell_f) = k \wedge \ell' \preceq \ell_f$

$\hookrightarrow$[def. of $\delta(\ell', \ell_f)$]

$\exists k > i, D \equiv_{Def} \{\ell_0, \ldots, \ell_k\}$.
(1) $D \subseteq R$, (2) D is a $\rightarrow$–chain with
first element $\ell_0 = \ell'$ and last element $\ell_k = \ell_f$

$\hookrightarrow [\ell \to \ell', \ell \neq \ell']$

$\exists \bar{D} \equiv_{Def} \{\ell, \ell_0, \ldots, \ell_k\}$.
(1) $D \subseteq \bar{D} \subseteq R$, (2) D is a $\to$–chain with
first element ℓ, $\ell \to \ell_0 = \ell'$, and last element $\ell_k = \ell_f$

$\hookrightarrow$[def. of $\Delta T(\ell)$] $\Delta T(\ell) \geq \delta(\ell, \ell_f) \geq |\bar{D}| - 1 = k + 1 > i + 1$

which contradicts the premise that $\ell \in T_{i+1}$;

$\hookrightarrow [\neg(\Delta T(\ell') > i)]$ $\Delta T(\ell') \leq i$

$\hookrightarrow$[def. of T_i] $\ell' \in T_i$.

The rest of the proof of theorem 3.3 is given in the appendix.

3.6.2 Stuttering invariant specifications

We next consider a particular subclass of specifications, which are not able to "count" steps (e.g. the number of cycles in a clocked device) [5].

The motivation for using this kind of specifications is that it abstracts from the "speed" of a computation. Instead of saying e.g.: "A request will be answered within N cycles", we would only require that a request will be granted *eventually*.

The next definition presents a subclass of Symbolic Automata (SA), which does not distinguish between stuttering–equivalent behaviours of a system.

Definition 3.17 (Stuttering–invariant SA) *Assume an assertion language $A\mathcal{L}$, and a set V of variables.*

A stuttering–invariant symbolic automaton (SA) $\mathcal{A}$ over V is an SA in normal form (cf. lemma 3.4)

$$A \equiv_{Def} (V, Locs, Edges, L_0, F)$$

where

$$\forall \ell_1, \ell_2 \,.\, \ell_1 \to \ell_2, \ell_1 \neq \ell_2 \;:\quad (\ell_1, \phi_1, \ell_2) \in Edges \wedge (\ell_2, \phi_2, \ell_2) \in Edges \implies \phi_1 \Rightarrow \phi_2$$

in words: The condition ϕ_1, which allows a transition from ℓ_1 to ℓ_2, must be

[5] The expert is referred in particular to a large body of work by Thomas Wilke for an in–depth coverage of the subject beyond the scope of this thesis; see e.g. [37] and other publications by Wilke and Wolper.

equivalent or stronger than the condition ϕ_2, which allows a loop at location ℓ_2.

The main consequence of this definition is stated in the next lemma.

Lemma 3.13 (Stuttering invariance) *Let $\mathcal{A}$ be a stuttering–invariant SA over some set of variables V.*

Let $\sigma \in L(\mathcal{A})$ be a computation over V which is accepted by $\mathcal{A}$.

Then:

$$\forall i \geq 0 \ : \ \sigma' \equiv_{Def} \sigma(0) \ldots \sigma(i-1)\sigma(i)\sigma(i)\sigma^{(i)} \in L(\mathcal{A})$$

i.e. a computation where step i is repeated, is also accepted by $\mathcal{A}$.

Proof of lemma 3.13 . The proof is trivial, because each transition in a run can be followed by an additional loop–step (by def. of stuttering invariance of $\mathcal{A}$).

Therefore, for the extended computation σ', there is an accepting run derived from the run which accepts the original computation σ.

q.e.d.

The next lemma indicates that a combination of the properties stuttering invariance and deterministic transition labels (def. 3.6) allows to eliminate the **next** –operator in certain LTL_V–specifications.

Lemma 3.14 (LTL–equivalences concerning next) *Assume an assertion language $A\mathcal{L}$, a set V of variables, LTL–formulas $\phi_0, \phi_1, \phi_2 \in LTL_V$ and define for $\mathcal{U} \in \{ \textbf{unless}, \textbf{until} \}$*

$$\begin{aligned} \phi_{\mathcal{U}} \quad &\equiv_{Def} \quad \phi_0 \wedge \textbf{next}\,(\phi_1 \,\mathcal{U}\, \phi_2) \\ \phi^{\circ}_{\mathcal{U}} \quad &\equiv_{Def} \quad \phi_0 \wedge (\phi_1 \,\mathcal{U}\, \phi_2) \quad . \end{aligned}$$

Then, for $\mathcal{U} \in \{ \textbf{unless}, \textbf{until} \}$:
If $\phi_0 \Rightarrow \phi_1$ then

$$\phi_{\mathcal{U}} \quad \Rightarrow \quad \phi^{\circ}_{\mathcal{U}} \quad ; \quad (1)$$

if $\phi_0 \Rightarrow \phi_1 \wedge \phi_1 \Rightarrow \neg\phi_2$ then:

$$\phi_{\mathcal{U}} \quad \sim \quad \phi^{\circ}_{\mathcal{U}} \quad . \quad (2)$$

The proof of this lemma is given in the appendix.

The next theorem shows that the semantics of a stuttering–invariant and deterministic POSA can be characterized by an LTL_V^- formula.

Theorem 3.4 (Translation from stuttering–invariant det. POSA to temporal logic) *Assume an assertion language $A\mathcal{L}$, and let $\mathcal{A}$ be a stuttering–invariant and deterministic POSA over some set V of variables. Then there exists a formula $\phi^\circ_{\mathcal{A}} \in LTL_V^-$ over V such that*

$$L(\mathcal{A}) = L(\phi^\circ_{\mathcal{A}}) \quad . \qquad (*)$$

Proof of theorem 3.4 – Construction. The construction is similar to the construction described in theorem 3.3.

Let $\mathcal{A} \equiv_{Def} (V, Locs, Edges, L_0, F)$. We construct $\phi^\circ_{\mathcal{A}}$ as follows:

For all locations $\ell \in Locs$, define

$$\phi^\circ_\ell \;=_{Def}\; \phi_{\ell,\ell} \quad \mathcal{U} \quad (\neg\phi_{\ell,\ell} \wedge (\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \phi^\circ_{\ell'}))$$

where

$$\begin{array}{lll} \mathcal{U} & =_{Def} \quad \textbf{unless} & \text{if } \ell \in F, \text{ and} \\ \mathcal{U} & =_{Def} \quad \textbf{until} & \text{if } \ell \notin F \end{array}$$

and let

$$\phi^\circ_{\mathcal{A}} \;=_{Def}\; \bigvee_{\ell\in L_0} \phi^\circ_\ell \quad .$$

Recall that a deterministic SA has one (unique) start location ℓ_0, so

$$\phi^\circ_{\mathcal{A}} \;=_{Def}\; \phi_{\ell_0}$$

We claim that under these definitions the following claims hold:

$$\forall \ell \;:\; \phi^\circ_\ell \sim \phi_\ell \quad (1)$$
$$\phi^\circ_{\mathcal{A}} \sim \phi_{\mathcal{A}} \quad (2)$$

where ϕ_ℓ is defined in theorem 3.3.

Since $L(\mathcal{A}) = L(\phi_{\mathcal{A}})$ (according to theorem 3.3), fact (2) implies (*).

Proof of claim (1). The proof is by induction, using the same monotonic sequence $(T_i)_{i\geq 0}$ of sets

$$T_i =_{Def} \{\ell \in R \mid \Delta T(\ell) \leq i\}$$

as defined in the proof of theorem 3.3 (where R is the set of reachable locations in $\mathcal{A}$; the main properties of the sequence $(T_i)_{i\geq 0}$ are stated in lemma 3.12).

case $\ell \in T_0$ (set of final locations). Recall that T_0 is *the set of maximal locations* with respect to the partial order $\preceq_{|R}$ (cf. lemma 3.12). For elements $\ell \in T_0$, no successor (different from ℓ) exists.

$\hookrightarrow[\ \bigvee_{i\in\emptyset} \phi_i \sim \mathbf{false}]\ \phi^{\circ}_{\ell} \sim (\phi_{\ell,\ell}\,\mathcal{U}\,\mathbf{false}) \sim \phi_{\ell}$

case $\ell \in T_{k+1}$, some $k\geq 0$. Assume that we have proven claim (1) for all i, $0\leq i\leq k$.

From the assumption made in the theorem – $\mathcal{A}$ is deterministic –, we get

$$\forall \ell', \ell \neq \ell', \ell \rightarrow \ell' \ :\ \phi_{\ell,\ell} \Rightarrow \neg\phi_{\ell,\ell'} \quad (1)$$

$$\hookrightarrow[\text{ def. of } \bigvee] \qquad \phi_{\ell,\ell} \Rightarrow \neg(\bigvee_{\ell\neq\ell' \,:\, \ell\rightarrow\ell'} \phi_{\ell,\ell'} \wedge \ldots) \quad (2)$$

$\hookrightarrow[$ definition of ϕ_ℓ in theorem 3.3$]$

$$\phi_\ell \equiv_{Def} \phi_{\ell,\ell} \quad \mathcal{U} \quad (\bigvee_{\ell\neq\ell' \,:\, \ell\rightarrow\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi_{\ell'})$$

$\hookrightarrow[$ (2)$]$

$$\phi_\ell \sim \phi_{\ell,\ell} \quad \mathcal{U} \quad (\neg\phi_{\ell,\ell} \wedge (\bigvee_{\ell\neq\ell' \,:\, \ell\rightarrow\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi_{\ell'}))$$

where $\ell' \in T_k$ for all successors ℓ' of ℓ.

$\hookrightarrow[$ induction premise: $\phi_{\ell'} \sim \phi^{\circ}_{\ell'}]$

$$\phi_\ell \sim \phi_{\ell,\ell} \quad \mathcal{U} \quad (\neg\phi_{\ell,\ell} \wedge (\bigvee_{\ell\neq\ell' \,:\, \ell\rightarrow\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi^{\circ}_{\ell'})) \quad (**)$$

From the assumption made in the theorem – $\mathcal{A}$ is stuttering invariant –, we have $\phi_{\ell,\ell'} \Rightarrow \phi_{\ell',\ell'}$ for successor locations ℓ, ℓ'; hence by lemma 3.14

$$
\begin{array}{lll}
(**) & \phi_\ell \sim \phi_{\ell,\ell} \quad \mathcal{U} \quad (\neg\phi_{\ell,\ell} \wedge (\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\, \phi^\circ_{\ell'})) \\
\sim & [\phi^\circ_{\ell'} =_{Def} \phi_{\ell',\ell'} \quad \mathcal{U} \quad (\neg\phi_{\ell',\ell'} \wedge \ldots)] \\
& \phi_{\ell,\ell} \quad \mathcal{U} \quad (\neg\phi_{\ell,\ell} \wedge (\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \phi^\circ_{\ell'})) \\
\sim & \phi^\circ_\ell \quad .
\end{array}
$$

Hence, claim (1) is proven.

Proof of claim (2). The claim is a corollary of claim (1): For the particular case of a deterministic POSA (with unique start location ℓ_0),

$\phi_{\mathcal{A}} = \phi_{\ell_0} \sim$[claim (1)] $\phi^\circ_{\ell_0} \sim \phi^\circ_{\mathcal{A}}$.

q.e.d.

3.6.3 Temporal logic characterization of deterministic POSA

This subsection is an application of the transformations described in theorem 3.1 and 3.2.

We will show that for a deterministic POSA, an equivalent characterization in $LINLTL_V$ (cf. def. 3.16) is possible.

Theorem 3.5 (Translation from deterministic POSA to sequencing temporal logic) *Assume an assertion language $A\mathcal{L}$, and let $\mathcal{A}$ be a deterministic POSA over some set V of variables. Then there exists a formula $\xi_{\mathcal{A}} \in LINLTL_V$ such that*

$$L(\mathcal{A}) = L(\xi_{\mathcal{A}}) \quad . \qquad (*)$$

Proof of theorem 3.5 . The proof is based on the construction of the formulas ϕ_ℓ given in theorem 3.3.

Note that $\mathcal{A}$ is deterministic and has therefore a unique initial location ℓ_0; hence $\phi_{\mathcal{A}} = \phi_{\ell_0}$.

We show that for each formula ϕ_ℓ, there is an equivalent formula $\xi_\ell \in LINLTL_V$, $\xi_\ell \sim \phi_\ell$. The proof is by induction, using the same monotonic sequence $(T_i)_{i\geq 0}$ of sets as defined in the proof of theorem 3.3 .

case $\ell \in T_0$ (set of final locations). For elements $\ell \in T_0$, no successor (different from ℓ) exists.

Define

$$\xi_\ell \equiv_{Def} \phi_{\ell,\ell}\,\mathcal{U}\,((\mathbf{false} \wedge \mathbf{false}) \vee \mathbf{false}) \in LINLTL_V$$

$$\hookrightarrow[\bigvee_{i\in\emptyset} \phi_i \sim \mathbf{false}]\ \phi_\ell \sim (\phi_{\ell,\ell}\,\mathcal{U}\,\mathbf{false}) \sim \xi_\ell$$

case $\ell \in T_{k+1}$, some $k{\geq}0$. We assume that for each $\ell \in T_k$, there exists a formula $\xi_\ell \in LINLTL_V$ such that $\xi_\ell \sim \phi_\ell$.

By definition,

$$\phi_\ell \equiv_{Def} \phi_{\ell,\ell} \quad \mathcal{U} \quad (\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi_{\ell'})$$

$\sim$[lemma 3.10, $\mathcal{A}$ det.]

$$\phi_{\ell,\ell} \quad \mathcal{U} \quad (\bigwedge_{\ell\neq\ell' \,:\, \ell\to\ell'} ((\phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi_{\ell'}) \vee \bar{\phi}_{\ell,\ell'}))$$

where

$$\bar{\phi}_{\ell,\ell'} \equiv_{Def} \bigvee_{\ell''\neq\ell',\ell\neq\ell'',\ell\to\ell''} \phi_{\ell,\ell''}$$

$\sim$[lemma 3.11, $\mathcal{A}$ det.]

$$\bigwedge_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi_{\ell'}) \vee \bar{\phi}_{\ell,\ell'})(*)$$

By induction hypothesis, we can assume that for $\phi_{\ell'}$ there is an equivalent formula $\xi_{\ell'} \in LINLTL_V$, which has the form

$$\xi_{\ell'} = \xi_1 \wedge \ldots \wedge \xi_{k(\ell')}$$

for some $k(\ell'){\geq}1$; for each $i = 1 \ldots k(\ell')$, ξ_i has the form

$$\xi_i \equiv_{Def} \beta_1^i\,\mathcal{U}\,((\beta_2^i \wedge \mathbf{next}\,\xi_1^i) \vee \beta_3^i) \quad .$$

Hence, (*) is equivalent to:

$\sim$[**next** distributes over conjunction]

$$\bigwedge_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \bigwedge_{i=1\ldots k(\ell')} \mathbf{next}\,\xi_i) \vee \bar{\phi}_{\ell,\ell'})$$

$\sim$[by lemma 3.11]

$$\bigwedge_{\ell \neq \ell' \,:\, \ell \rightarrow \ell'} \; \bigwedge_{i=1 \ldots k(\ell')} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \mathbf{next}\, \xi_i) \vee \bar{\phi}_{\ell,\ell'}) \equiv_{Def} \xi_\ell \quad .$$

With this definition, $\xi_\ell \in \mathit{LINLTL}_V$, which concludes the proof.

q.e.d.

3.7 Summary

This chapter has introduced the theoretical basis for the semantics definition of the graphical specification language STD.

It has been shown that for a certain subclass of automata with Büchi–fairness condition (POSA), a corresponding characterization in LTL (LTL_V) can be found.

Furthermore, an interesting subclass of LTL_V, LTL_V^-, has been introduced. In this chapter we focused on the property that for each formula $\phi \in \mathit{LTL}_V^-$, there exists an equivalent formula in the – even more restricted – logic LINLTL_V. The logic LINLTL_V will provide the basis for the semantics definition of LSTD (see chapter 5).

Finally, a certain subclass of POSA – *deterministic* POSA – has been considered. It has been shown that for this type of SA, a characterization in LINLTL_V can be given.

Deterministic POSA will play an important role in chapter 6 to describe the semantics of an STD body. In chapter 5, it will be shown that the semantics of deterministic POSA can be characterized in the framework of LSTD, which can be considered as a graphical variant of the sequencing logic LINLTL_V.

Chapter 4

Theoretical foundation of model construction

In this chapter, we introduce a formal framework for the definition of implementation models.

In chapter 2, we mentioned that the verification environment for Statemate uses an intermediate language called SMI to represent models. Whereas SMI is a full–fledged language, the model which we will use here is much simpler. Yet, it is sufficiently rich to express our demands — namely, the definition of models for synchronous systems. The baseline for the exposition in this chapter is mainly adopted from the approach presented in [26].

This chapter introduces the following items:

- Fair transition systems
- Transition graphs, and
- transition graph systems (TGS)
- open TGS
- module composition, and
- compositional reasoning.

4.1 Fair transition systems

In this chapter we introduce a formalism for the representation of a system model $\mathcal{M}$. The formalism is adopted from [26] and called "fair transition system" (FTS).

FTS is a formalism to describe *generators* of computations. As has been demonstrated in [26], this basic formalism is well suited to serve as a model for operational semantics definitions of more complicated operational specification languages using various constructs of concurrent programming languages (e.g. parallel processes, synchronous and asynchronous communication etc.).

4.1.1 Structure and semantics of FTS

The next definition defines the structure of fair transition systems.

Definition 4.1 (Fair transition system) *Assume a set of variables $\mathcal{V}$, and an assertion language $A\mathcal{L}$. A fair transition system TS over V is a structure*

$$(V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}) \qquad \textit{(FTS)}$$

with the following components:

- *$V \subseteq \mathcal{V}$ is a finite set of state variables,*
- *$\Theta \in A\mathcal{L}$ is an initial condition, which is satisfiable; i.e., $\phi_0 =_{Def} \langle\!|\Theta|\!\rangle$ is satisfiable in the sense of temporal logic ($L(\phi_0) \neq \emptyset$);*
- *$\mathcal{T}$ is a finite set of transitions. A transition $\tau \in \mathcal{T}$ is a mapping*

$$\tau \ : \ \rho \in \Sigma \mapsto R'_\rho \subseteq \Sigma \quad ,$$

where $\Sigma =_{Def} \mathrm{Val}(V)$ is the set of valuations of the variables in V. It is required that $\mathcal{T}$ always contains one particular transition τ_I, called idling transition, defined by $\tau_I =_{Def} (\rho \mapsto \{\rho\})$.

- *$\mathcal{J} \subseteq \mathcal{T}$ is set of weakly fair (or "just") transitions;*
- *$\mathcal{C} \subseteq \mathcal{T}$ is set of strongly fair (or "compassionate") transitions.*

The definition of a transition in definition 4.1 is rather abstract. Since for a transition $\tau \in \mathcal{T}$, $\tau(\rho) \subseteq \Sigma$ is a set with possibly more than one value, "taking" transition τ in state ρ means a *nondeterministic* transition to one successor state $\rho' \in \tau(\rho)$. In the special case that $\tau(\rho)$ is empty ($\tau(\rho) = \emptyset$) in state ρ, transition τ is said to be *disabled* at ρ; otherwise ($\tau(\rho) \neq \emptyset$), τ is said to be *enabled* at ρ.

The name "fair transition system" indicates that the formalism FTS combines two different aspects: (1) the notion of a (state–)transition system (STS), and (2) the notion of *fairness*.

Fairness plays an important role in the theory of concurrent programming languages as "abstract" model of a scheduler, and has been a dedicated research topic ([15]).

In FTS, two different notions of fairness are supported. Informally, a transition $\tau \in \mathcal{J}$ disallows a computation where τ is continously enabled from some point on but not taken (justice requirement). A transition $\tau \in \mathcal{C}$ disallows a computation where τ is infinitely often enabled, but only taken a finite number of times (compassion requirement).

The semantics of a FTS TS over a set of variables V is a set of computations over V, denoted as $L(TS)$. The next definition defines when a computation $\sigma \in Comp(V)$ is generated by TS (i.e., $\sigma \in L(TS)$).

Definition 4.2 (Computations of FTS) *Let TS be a FTS over a set of variables V. A computation*

$$\sigma = (t \in N_0 \mapsto \rho_t \in Val(V)) \in Comp(V)$$

over V is a computation of TS ($\sigma \in L(TS)$), iff the following conditions hold:

- *Initiation: The first state of the computation σ (ρ_0) is initial, i.e.$\rho_0 \models \langle\!|\Theta|\!\rangle$.*
- *Consecution: For each pair of consecutive states ρ_t, ρ_{t+1} in σ, there is a transition $\tau \in \mathcal{T}$ such that $\rho_{t+1} \in \tau(\rho_t)$ (ρ_{t+1} is a τ–successor of ρ_t). We refer to the pair ρ_t, ρ_{t+1} as a τ–step, and say that τ is taken at moment t. Note that it is possible for a given pair ρ_t, ρ_{t+1} to be both a τ–step and a τ'–step for some τ', $\tau \neq \tau'$.*
- *Justice: For each transition $\tau \in \mathcal{J}$ it is not allowed that τ is continually enabled in σ from some moment t on, but never taken after t.*
- *Compassion: For each transition $\tau \in \mathcal{C}$ it is not allowed that τ is enabled at infinitely many moments, but only taken at a finite number of moments.*

4.2 Transition graph systems

The FTS formalism is a semantic model rather than a language. Next we will introduce the notion of a "transition graph system" as simple operational language which is appropriate for the specification of non–terminating concurrent programs, based on an interleaving semantics.

A *transition graph system* consists of two parts: (1) a declarative part, and (2) a "concurrent process" definition part. First, we introduce a diagrammatic method for the specification of processes, called "transition graphs".

Definition 4.3 (Transition Graph) *Assume a set $\mathcal{V}$ of variables and an assertion language $A\mathcal{L}$. A transition graph (TG) is a directed graph*

$$\mathcal{G} \ : \ (Locs, Edges, \ell_0, E\bar{\mathcal{J}}, E\mathcal{C}) \quad ,$$

where

- *$Locs \subseteq \mathcal{D}^{Locs}$ is a finite set of nodes (referred to as locations) taken from a designated set $\mathcal{D}^{Locs}$ of control locations, and*
- *Edges is a set of edges. Edges are triples*

$$(\ell, label, \ell')$$

where ℓ, ℓ' are control locations and label is an (atomic) instruction, which has the form of a guarded (multiple) assignment

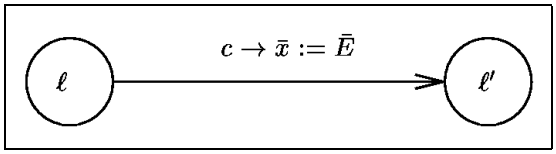
$$c \rightarrow \bar{x} := \bar{E}$$

where $c \in A\mathcal{L}$ is an assertion called the guard of the instruction, $\bar{x} \equiv [x_1, \ldots, x_k]$ is a list of variables with

$$x_i \in \mathcal{V} \quad ,$$

and $\bar{E} \equiv [E_1, \ldots, E_k]$ is a list of expressions, such that x_i and E_i have the same type ($i = 1 \ldots k$).

Figure 4.1: Diagrammatic presentation of transition.



- $\ell_0 \in Locs$ *is the initial control location.*
- $E\bar{\mathcal{J}} \subseteq Edges$ *is a set of edges which are not subject to the requirement of justice;*
- $E\mathcal{C} \subseteq Edges \backslash E\bar{\mathcal{J}}$ *is a set of edges which are subject to the requirement of compassion. The semantics of these sets will be explained in definition 4.5 below.*

We allow the following abbreviations in a transition graph: The empty assignment $[] := []$ is abbreviated as skip; instead of $c \to \mathbf{skip}$ we write $c?$. Another special case is $c = \mathbf{true}$; instead of $\mathbf{true} \to \bar{x} := \bar{E}$ we write $\bar{x} := \bar{E}$.

In a diagrammatic presentation of the transition graph, locations are represented by labelled circles, and an edge $e \in Edges$,

$$e \equiv_{Def} (\ell, c \to \bar{x} := \bar{E}, \ell')$$

is represented by a labelled arrow between the circles representing the source and destination, as shown in figure 4.1.

If $e \in E\bar{\mathcal{J}}$, then the arrow–label *label* is enclosed in square brackets $[label]$; if $e \in E\mathcal{C}$, then it is denoted as $\gg label \ll$. The initial location is represented by a circle with a (dangling) arrow pointing to it.

The notion of a transition graph system is formalized in next definition.

Definition 4.4 (Transition Graph System) *Assume a set $\mathcal{V}$ of variables and an assertion language $A\mathcal{L}$. A transition graph system (TGS) has the following form:*

$$\mathcal{GS} \quad : \quad \textbf{program} \quad \langle\text{program-name}\rangle$$
$$\langle\text{interface-declaration}\rangle$$
$$\mathcal{G}_1 \| \ldots \| \mathcal{G}_m$$

where

- $\mathcal{G}_1, \ldots, \mathcal{G}_m$, $m \geq 1$ *are transition graphs defining a set of concurrent processes; the operator* $\|$ *denotes the (interleaved) concurrent composition of these processes. W.l.o.g. we assume that the sets of control–locations* $Locs_{\mathcal{G}_i}$ *of the respective transition graphs are pairwise disjoint.*
- $\langle$interface-declaration$\rangle$ *defines a set* $V_{\mathcal{GS}} \subseteq \mathcal{V}$ *of typed variables accessible to all concurrent processes for reference and modification. An interface–declaration consists of a sequence of interface–declaration–statements of the form*

$$\langle\text{mode}\rangle \quad \langle var_1\rangle, \ldots, \langle var_k\rangle : \langle\text{type}\rangle \quad [\textbf{where}\langle\text{init-cond}\rangle]$$

where the token $\langle$mode$\rangle$ *of each declaration statement is defined to be either* **in**, **out** *or* **local**, $x \equiv_{Def} \langle\text{var}\rangle$ *declares a variable* $x \in V^{\tau}$ *with* $\tau \equiv_{Def} \langle\text{type}\rangle$, *and the optional "where" part contains with* $E \equiv_{Def} \langle\text{init-cond}\rangle$ *a so–called initialization–assertion* $E \in A\mathcal{L}$.

A number of well–formedness conditions are imposed on a TGS. We call a TGS *well–formed*, if the following conditions are met:

1. For each transition labelled by the multiple assignment statement
$$c \rightarrow \bar{x} := \bar{E}$$
all variables in $\bar{x}$ must be of mode **out** or **local**;
2. for an interface–declaration–statement of mode **in**, the initialization assertion refers only to variables declared with mode **in**;
3. for an interface–declaration–statement of mode **out** or **local**, the initialization–assertion must be of the form
$$x_1 = E_1 \textbf{ and } \ldots \textbf{ and } x_n = E_n$$
where x_i is a variable declared in that statement, and E_i is an expression that refers only to variables declared with mode **in**, for $i = 1 \ldots n$.

In the following we will assume that a TGS is well–formed.

As can be seen from the well–formedness conditions stated above, the modes of the interface–declaration play an important role for the static, and also (as will be demonstrated later) for the behavioral semantics. Only variables of mode **in** and **out** are considered to be "observable" from the concurrent program specified by the TGS. Variables of mode **in** can be initialized externally once, viz. at the beginning of a computation; the values for these variables must conform to the associated initialization assertion (if present).

Thus the program may run under various initial conditions. Variables of mode **local** are considered to be "hidden" from an external observer; in particular, an abstract specification of the program cannot refer to (the values of) local variables.

4.2.1 Semantics of transition graph systems

We have already stated that the formalism FTS introduced in definition 4.1 serves as a means for operational semantics definition. We now show how the semantics of the simple language TGS can be formalized using FTS.

The key concept of this semantics definition is that concurrency is modelled by "interleaving" with additional fairness constraints. This model of concurrency is especially appropriate w.r.t. verification. We refer the reader to [26] for a detailed discussion of this approach.

The next definition defines the semantics of TGS.

Definition 4.5 (Semantics of TGS) *Assume a set $\mathcal{V}$ of variables, an assertion language $A\mathcal{L}$, and a transition graph system $\mathcal{GS}$, whose concurrent process definition part is*

$$\mathcal{G}_1 \| \ldots \| \mathcal{G}_m \quad ;$$

let $\mathcal{G}_i \equiv_{Def} (Locs_i, Edges_i, \ell_{0,i}, E\bar{\mathcal{J}}_i, E\mathcal{C}_i)$, for $i = 1 \ldots m$. Then the semantics of $\mathcal{GS}$, denoted $L(\mathcal{GS})$, is defined to be the set of computations of the FTS

$$TS_{\mathcal{GS}} \equiv_{Def} (V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$$

The components of $TS_{\mathcal{GS}}$ are defined as follows:

- $V =_{Def} V_{\mathcal{GS}} \dot{\cup} \{\pi_1, \ldots, \pi_m\}$, *where each of the fresh variables π_i, $\{\pi_1, \ldots, \pi_m\} \subseteq \mathcal{V}^{Locs}$, taken from a designated variable set $\mathcal{V}^{Locs}$ disjoint from $\mathcal{V}$, represents the current control location in the TG $\mathcal{G}_i$.*

- $\Theta =_{Def} (\mathbf{and}_{i=1}^{r} IA_i)$ **and** $(\mathbf{and}_{j=1}^{m}(\pi_j = \ell_{0,j}))$, *where* IA_i *is the initialization assertion of the* i*–th out of* r *interface declaration statements in* $\mathcal{GS}$*, and* $\ell_{0,j}$ *the initial location of the* j*–th transition graph in* $\mathcal{GS}$*.*

- $\mathcal{T}$ *is the set of transitions defined by the following rule (R): For each labelled edge*

$$e \equiv_{Def} (\ell_1, c \to x_1, \ldots, x_k := E_1, \ldots, E_k, \ell_2),$$

$e \in Edges_i$ and $k \geq 0$, $\mathcal{T}$ contains a transition τ_e, defined by the extended guarded assertion over V

$$\begin{array}{lll} \tau_e =_{Def} (\rho & \mapsto & \emptyset, \quad \textit{if } [\![(\pi_i = \ell_1) \ \mathbf{and} \ c]\!]\rho = \mathbf{false} \quad ; \\ & \mapsto & \{\rho' \in Val(V) \mid (\rho(\pi_i) = \ell_2) \textit{ and } \rho'(x_i) = [\![E_i]\!]\rho, i = 1 \ldots k, \\ & & \quad \textit{and } \forall y \in V \backslash \{x_1, \ldots, x_k\} \,.\, \rho'(y) = \rho(y)\} \quad , \\ & & \quad \textit{if } [\![(\pi_i = \ell_1) \ \mathbf{and} \ c]\!]\rho = \mathbf{true} \quad). \end{array}$$

The set $\mathcal{T}$ contains exactly the transitions defined by rule (R), plus one special state–preserving transition τ_I called idling transition, defined by

$$\tau_I : \rho \in Val(V) \mapsto \{\rho\} \quad .$$

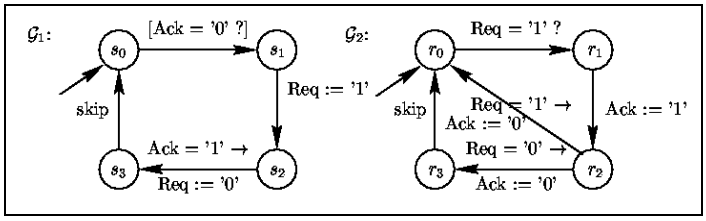
- $\mathcal{J} =_{Def} \{\tau_e \mid e \in \bigcup_{i=1}^{m} Edges_i \backslash E\bar{\mathcal{J}}_i\}$ *is the set of transitions subject to the requirement of justice; and*

- $\mathcal{C} =_{Def} \{\tau_e \mid e \in \bigcup_{i=1}^{m} E\mathcal{C}_i\}$ *is the set of transitions subject to the requirement of compassion.*

"Interleaving" means that the execution of a concurrent program consists of an (infinite) sequence of *interleaved actions* (state transitions) of its processes. The fairness conditions ensure, that the interleaving is "fair", e.g. that each process is scheduled infinitely often.

The "standard" fairness condition defined by the semantics of TGS is the requirement of justice, which is attributed to each transition representing an edge in one of the transition graphs of the TGS, unless it is excluded explicitly from this requirement (via the set $E\bar{\mathcal{J}}$ of the TGS).

Figure 4.2: TGS model of 4–phase handshake protocol.



The possibility to release the fairness for a transition allows a simple modeling of non–deterministic behavior. On the other hand, the possibility to attribute the stronger requirement of compassion to a transition representing an edge allows to strengthen the fairness requirement (via the set $\mathcal{EC}$ of the TGS) in order to handle situations where more than one transitions leave from the same control location.

Example 4.1 (Req/Ack–handshake protocol) *We consider a simple version of a 4–phase asynchronous handshake protocol, assuming a type Bit with value domain* $\mathcal{D}^{Bit} =_{Def} \{\text{'0'}, \text{'1'}\}$.

We define a TGS as follows:

$$\begin{aligned} \mathcal{GS}_{Req,Ack} : \quad & \textbf{program } \textit{Req\_Ack\_Protocol} \\ & \textbf{out } \text{Req} : \text{Bit } \textbf{where } \text{Req} = \text{'0'} \\ & \textbf{out } \text{Ack} : \text{Bit } \textbf{where } \text{Ack} = \text{'0'} \\ & \mathcal{G}_1 \| \mathcal{G}_2 \end{aligned}$$

where $\mathcal{G}_1$ *represent the model of a master responsible for setting the Req–signal, and* $\mathcal{G}_2$ *represents the model of a slave responsible for setting the Ack–signal. The slave can reset the Ack–signal while the Req–signal is active (Req = '1'). $\mathcal{G}_1, \mathcal{G}_2$ are defined in the diagrammatic presentation of the graphs shown in figure 4.2.*

4.2.2 Verification of properties of a TGS

We discuss possible computations of the TGS $\mathcal{GS} \equiv_{Def} \mathcal{GS}_{Req,Ack}$ shown in example 4.1. We use the following abbreviations for the defined edges (where _ denotes the unique label of the corresponding TGS transition):

$$e^s_{ij} =_{Def} (s_i, \_, s_j) \text{ and } e^r_{ij} =_{Def} (r_i, \_, r_j) \quad ;$$

the corresponding transitions induced in the FTS $TS \equiv_{Def} TS_{\mathcal{GS}}$ are abbreviated as τ^s_{ij} and τ^r_{ij}, respectively.

For a TGS $\mathcal{GS}$ and temporal logic formula ϕ, define the *correctness assertion* $\mathcal{GS} \models \phi$ by

$$\mathcal{GS} \models \phi \textbf{ iff } L(\mathcal{GS}) \subseteq L(\phi) \quad .$$

We investigate if the following properties hold for all computations of GS.

> Property 1: "(In all computations,) *Req* will be eventually asserted."

This property is described by the temporal logic formula

$$\phi_1 =_{Def} \textbf{eventually} \langle\!| Req = \text{'1'} |\!\rangle \quad ;$$

Does the correctness assertion $\mathcal{GS} \models \phi_1$ hold ?

The answer is *no*. First note that the set of variables of the FTS TS is $V_{TS} = \{$ Req , Ack $, \pi_1, \pi_2\}$. Each computation starts with the unique initial state ρ_0 for which

$$[\![Req = \text{'0'} \textbf{ and } Ack = \text{'0'} \textbf{ and } \pi_1 = s_0 \textbf{ and } \pi_2 = r_0]\!] \rho_0 = \textbf{true} \quad .$$

In state ρ_0, exactly two transitions are enabled: (1) transition τ^s_{01}, and (2) the state–preserving *idling transition* τ_I. Since $\tau^s_{01} \in \bar{\mathcal{J}}_1$, this transition is excluded from the requirement of justice. Therefore a computation exists where the idling transition τ_I is taken infinitely often, and the state remains unchanged. In particular, no state ρ is ever reached such that $[\![Req = \text{'1'}]\!] \rho =$ **true** holds.

> Property 2: "(In all computations,) whenever *Req* is asserted, then it will be de–asserted eventually thereafter."

This property is described by the formula

$$\phi_2 =_{Def} \textbf{always} \, (\langle\!| Req = \text{'1'} |\!\rangle \rightarrow \textbf{eventually} \langle\!| Req = \text{'0'} |\!\rangle) \quad ;$$

Does the correctness assertion $\mathcal{G}S \models \phi_2$ hold ?

Again, the answer is *no*. It is possible to find a computation which violates ϕ_2, i.e. one which satisfies $\neg\phi_2$, which is

$$\neg\phi_2 \sim \textbf{eventually}\,(\langle\!|Req = \textit{'1'}|\!\rangle \wedge \textbf{always}\,\neg\langle\!|Req = \textit{'0'}|\!\rangle) \quad ;$$

$\neg\phi_2$ can be read as: "Eventually a state is reached where *Req* is asserted and is never de–asserted thereafter."

One computation which satisfies $\neg\phi_2$ is the state–sequence $\sigma \equiv_{Def} (\rho_i)_{i=0}^{\infty}$ resulting from the following sequence $\sigma\tau$ of transitions:

$$\sigma\tau \equiv_{Def} \tau_{01}^s\tau_{12}^s(\tau_{01}^r\tau_{12}^r\tau_{20}^r)^\omega \quad ,$$

where $(\ldots)^\omega$ means the infinite repetition of $\ldots$. After taking the first two transitions of this sequence, state ρ_2 is reached, for which

$$[\![Req = \textit{'1'}\ \textbf{and}\ Ack = \textit{'0'}\ \textbf{and}\ \pi_1 = s_2\ \textbf{and}\ \pi_2 = r_0]\!]\rho_2 = \textbf{true} \quad .$$

For the rest of the transition sequence $(\tau_{01}^r\tau_{12}^r\tau_{20}^r)^\omega$, each of the resulting state ρ_i, $i > 2$, satisfies

$$[\![Req = \textit{'1'}]\!]\rho_i = \textbf{true} \quad .$$

Obviously the state sequence (computation) σ satisfies $\neg\phi_2$; the remaining question is whether the transition sequence respects the fairness constraints. In particular, consider transition τ_{23}^s, which is enabled and disabled periodically during the transition sequence $(\tau_{01}^r\tau_{12}^r\tau_{20}^r)^\omega$; but since it is not enabled from some moment on *constantly*, the justice requirement does not apply.

The situation becomes different, if edge e_{23}^s is included into the compassion set $\mathcal{C}_1$ of the transition graph $\mathcal{G}_1$. Then, the requirement of compassion is violated for the transition sequence $\sigma\tau$, where transition τ_{23}^s is enabled infinitely often, but not taken infinitely often. If this modification ($\mathcal{C}_1 =_{Def} \{e_{23}^s\}$) is applied to the TGS $\mathcal{G}S$, then property 2 holds.

4.3 Modules and composition

The model of transition graph systems introduced in the previous section is used to describe non–terminating concurrent programs. These programs represent models of "closed" systems, where no external interaction is possible

during the runs (computations) of the program. The only possibility of an external influence is during the choice of initial values for variables of mode **in**. Note that this choice may be restricted by the program via the initialization condition.

In order to model *reactive* systems – i.e., systems whose behavior is determined by continous interaction with their environment – we will next generalize the notion of TGS.

4.3.1 Open Transition Graphs Systems

The notion of an "open" transition graph system is formalized in next definition.

Definition 4.6 (Open Transition Graph System) *Assume a set $\mathcal{V}$ of variables and an assertion language $A\mathcal{L}$. An open transition graph system (OTGS) has the following form:*

$$\begin{array}{rl} \mathcal{G}M\ : & \textbf{module}\quad \langle\text{module-name}\rangle \\ & \langle\text{interface-declaration}\rangle \\ & \mathcal{G}_1 \| \ldots \| \mathcal{G}_m \end{array}$$

where

- $\mathcal{G}_1, \ldots, \mathcal{G}_m$, $m \geq 1$ *are transition graphs defining a set of reactive concurrent processes; the operator* $\|$ *denotes the (interleaved) concurrent composition of these processes. W.l.o.g. we assume that the sets of control–locations* $Locs_{\mathcal{G}_i}$ *of the respective transition graphs are pairwise disjoint.*

- $\langle$interface-declaration$\rangle$ *defines a set* $V_{\mathcal{G}M} \subseteq \mathcal{V}$ *of typed variables accessible to all processes and the system environment for reference and modification. An interface–declaration consists of a sequence of interface–declaration–statements of the form*

$$\langle\text{mode}\rangle \quad \langle var_1\rangle, \ldots, \langle var_k\rangle : \langle\text{type}\rangle \quad [\textbf{where}\ \langle\text{init-cond}\rangle]$$

where the token $\langle$mode$\rangle$ *of each declaration statement is defined to be either* **in**, **out** *or* **local** *(as for TGS), and in addition* **external**. $x \equiv_{Def}$ $\langle$var$\rangle$ *declares a variable* $x \in V^\tau$ *with* $\tau \equiv_{Def}$ $\langle$type$\rangle$ *, and the optional*

"where" part contains with $E \equiv_{Def}$ ⟨init-cond⟩ *a so–called initialization–assertion* $E \in A\mathcal{L}$. *For variables of mode* **external**, *no initialization–assertion is allowed.*

A number of well–formedness conditions are imposed on a OTGS similar as done for TGS. We call a OTGS *well–formed*, if the following conditions are met:

1. For each transition labelled by the statement

$$c \rightarrow \bar{x} := \bar{E}$$

all variables in $\bar{x}$ must be of mode **out** or **local**;

2. for an interface–declaration–statement of mode **in** , the initialization assertion refers only to variables declared with mode **in**;

3. for an interface–declaration–statement of mode **out** or **local**, the initialization–assertion must be of the form

$$x_1 = E_1 \textbf{ and } \ldots \textbf{ and } x_n = E_n$$

where x_i is a variable declared in that statement, and E_i is an expression that refers only to variables declared with mode **in**, for $i = 1 \ldots n$.

In the following we will assume that an OTGS is well–formed.

The new concept of variables of mode **external** is best illustrated by the definition of the formal semantics of OTGS.

Definition 4.7 (Semantics of OTGS) *Assume a set* $\mathcal{V}$ *of variables, an assertion language* $A\mathcal{L}$, *and a transition graph system* $\mathcal{G}M$. *Then the semantics of* $\mathcal{G}M$, *denoted* $L(\mathcal{G}M)$, *is defined to be the set of computations of the FTS*

$$TS_{\mathcal{G}M} \equiv_{Def} (V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$$

where the components of $TS_{\mathcal{G}M}$ *are defined as for TGS (cf. def. 4.5), which one modification: The set of transitions* $\mathcal{T}$ *is augmented by an additional transition* τ_E *which represents a transition of the system environment. Such a transition may modify only variables of mode* **external**; *we denote this set of variables by* V_E *and define*

$$\tau_E \ : \ \rho \in \mathit{Val}(V) \mapsto \{\rho' \in \mathit{Val}(V) \ | \forall x \in V \backslash V_E \ . \ \rho'(x) = \rho(x)\} \quad .$$

Comparison of OTGS versus VHDL. It is interesting to note the relationship of the concept of variable modes in OTGS versus port modes in VHDL. (1) The mode **in** in OTGS means, that a variable can be set only once, at the beginning of a computation; this corresponds to the notion of *generics* (system parameters) in VHDL. (2) The mode **external** in OTGS means, that a variable can be changed by the environment infinitely often during a system computation; this corresponds to the port mode **in** in VHDL. (3) The mode **out** in OTGS defines a variable which can be read inside the system, and is also visible for the system environment; this corresponds to the notion of *buffer ports* in VHDL. Finally, (4) the mode **local** in OTGS models a "private" variable of the system, which is hidden from the environment; this corresponds to the notion of *signals* in VHDL.

4.3.2 Module composition

The notion of concurrent programs as modelled by TGS relies on a parallel composition of processes, which communicate via (shared) variables declared in the program interface. Similarly, modules (modelled by OTGS) can be composed to form more complex modules or (closed) concurrent programs.

First we define the notion of interface compatibility with respect to parallel composition of modules.

Definition 4.8 (interface compatibility w.r.t. parallel composition) *Assume two OTGS $\mathcal{G}M_1$, $\mathcal{G}M_2$ with respective interface declarations (ID) ID_1, ID_2. The interface declarations ID_1 and ID_2 are said to be compatible with respect to parallel composition, if the following conditions hold:*

- *(compatible modes) a variable x, which is declared both in ID_1 and ID_2, has either mode* **external** *or mode* **in** *in both interface declarations, or mode* **external** *in one and mode* **out** *in the other interface declaration;*

- *(compatible initialization) for a variable x, which is declared both in ID_1 and ID_2 with mode* **in** *and initialization–assertions IA_1 and IA_2, respectively, IA_1* **and** *IA_2 must be satisfiable under some initial state ρ_0, i.e. the following must hold:*

$$\exists \rho_0 \,.\, [\![IA_1 \textbf{ and } IA_2]\!]\rho_0 = \textbf{true} \quad .$$

Two modules – referred to as (module) "component" – can be composed to form a more complex module, if their interfaces are compatible in the sense of definition 4.8.

The interface of a composed module is mainly determined by the component interfaces; the only choice concerns the visibility of some of the declared variables, i.e. a choice can be made to declare a variable of mode **out** in a component to have either mode **out** or mode **local** in the composition. In the latter case, the variable is *hidden* by the composition.

Definition 4.9 (Module composition) *Assume two OTGS* $\mathcal{G}M_1$, $\mathcal{G}M_2$ *of the form*

$$\begin{array}{rl} \mathcal{G}M_i \ : & \textbf{module} \quad \langle \text{module-name}_i \rangle \\ & \langle \text{interface-declaration}_i \rangle \\ & \mathcal{G}^i_1 \| \ldots \| \mathcal{G}^i_{m^i} \end{array}$$

for $i = 1, 2$ *with compatible interface declarations* $ID_i \equiv_{Def} \langle \text{interface-declaration}_i \rangle$ *. W.l.o.g. we assume that the (union–)sets of control locations defined in the transition graphs of* $\mathcal{G}M_1$ *and* $\mathcal{G}M_2$ *are disjoint.*

Then the OTGS $\mathcal{G}M$ *is a composition of* $\mathcal{G}M_1$ *and* $\mathcal{G}M_2$*, denoted by*

$$\mathcal{G}M \ : \ \mathcal{G}M_1 \| \mathcal{G}M_2 \quad ,$$

if it has the form

$$\begin{array}{rl} \mathcal{G}M \ : & \textbf{module} \quad \langle \text{module-name} \rangle \\ & \langle \text{interface-declaration} \rangle \\ & \mathcal{G}^1_1 \| \ldots \| \mathcal{G}^1_{m^1} \quad \| \quad \mathcal{G}^2_1 \| \ldots \| \mathcal{G}^2_{m^2} \end{array}$$

and the interface declaration $ID \equiv_{Def} \langle \text{interface-declaration} \rangle$ *obeys to the following rules:*

- *R–ext/ext: If a variable* x *is declared both in* ID_1 *and* ID_2 *with mode* **external***, then it is declared in ID with mode* **external***.*
- *R–in/in: If a variable* x *is declared both in* ID_1 *and* ID_2 *with mode* **in** *and initialization–assertions* IA_1 *and* IA_2*, respectively, then it is declared in ID with mode* **in** *and initialization–assertions* IA_1 **and** IA_2*.*

- *R–out/ext: If a variable x is declared in ID_i with mode* **out** *and initialization–assertion IA_i, and in $ID_{\bar{\imath}}$ with mode* **external**, *for $i \in \{1,2\}, \bar{\imath} \in \{1,2\}\backslash\{i\}$, then it is declared in ID with mode* **out** *or mode* **local** *and initialization–assertion IA_i.*
- *R–unique: If a variable x is declared in exactly one interface with mode* **external**, **in** *or* **out** *with initialization–assertion IA(let $IA \equiv_{Def}$* **true** *if no IA is given), then x is declared in ID with the same mode and the same initialization–assertion IA.*
- *R–max: The interface declaration ID contains only variable declarations induced by one of the rules R–ext/ext, R–in/in, R–out/ext or R–unique.*

In order to illustrate definition 4.9 of module composition, we consider an equivalent variant of example 4.1.

Example 4.2 (Distributed Req/Ack–handshake protocol) *Assume the definition of the following OTGS :*

$$\begin{aligned} \mathcal{G}M_{Req} \; : \quad & \textbf{module} \;\; \text{Master} \\ & \textbf{out} \;\; \text{Req} : \text{Bit} \; \textbf{where} \; \text{Req} = \text{'0'} \\ & \textbf{external} \;\; \text{Ack} : \text{Bit} \\ & \mathcal{G}_1 \end{aligned}$$

and

$$\begin{aligned} \mathcal{G}M_{Ack} \; : \quad & \textbf{module} \;\; \text{Slave} \\ & \textbf{out} \;\; \text{Ack} : \text{Bit} \; \textbf{where} \; \text{Ack} = \text{'0'} \\ & \textbf{external} \;\; \text{Req: Bit} \\ & \mathcal{G}_2 \end{aligned}$$

where $\mathcal{G}_1$ and $\mathcal{G}_2$ are defined as in example 4.1. Then

$$\begin{aligned} \mathcal{G}M_{Req,Ack} \; : \quad & \textbf{module} \; \textit{Distributed\_Req\_Ack\_Protocol} \\ & \textbf{out} \;\; \text{Req} : \text{Bit} \; \textbf{where} \; \text{Req} = \text{'0'} \\ & \textbf{out} \;\; \text{Ack} : \text{Bit} \; \textbf{where} \; \text{Ack} = \text{'0'} \\ & \mathcal{G}_1 \| \mathcal{G}_2 \end{aligned}$$

is a composition of $\mathcal{G}M_{Req}$ *and* $\mathcal{G}M_{Ack}$.

It is easy to see that for the special case of an OTGS $\mathcal{G}M$ with *no* variables of mode **external**, the OTGS can be transformed into an TGS $\mathcal{G}S$, replacing keyword **module** by keyword **program**, without changing the semantics: The additional environment transition τ_E in the FTS $TS_{\mathcal{G}M}$ generated from $\mathcal{G}M$ is in this case the same as the idling transition τ_I. Omitting the redundant transition τ_E gives the FTS $TS_{\mathcal{G}S}$ generated from $\mathcal{G}S$ as result; hence $L(\mathcal{G}M) = L(\mathcal{G}S)$. In particular, the module *Distributed_Req_Ack_Protocol* in example 4.2 is equivalent to program *Req_Ack_protocol* in example 4.1.

4.4 Basis for compositional reasoning

An important aspect of the semantics of modules (OTGS) with respect to verification is the question, whether a property proven of a module remains valid, if this module is composed with another module (property "preservation" under composition). More general, we would like to establish the following rule

R–comp:

$$\frac{\mathcal{G}M_1 \models \phi_1, \quad \mathcal{G}M_2 \models \phi_2, \quad \mathcal{G}M \ : \ \mathcal{G}M_1 \| \mathcal{G}M_2}{\mathcal{G}M \models \phi_1 \wedge \phi_2}$$

where $\mathcal{G}M_1$, $\mathcal{G}M_2$ are OTGS, $\mathcal{G}M$ is a composition of $\mathcal{G}M_1$ and $\mathcal{G}M_2$, and ϕ_1,ϕ_2 are temporal logic formulas.

Equally important is the reverse interpretation of rule R–comp: In order to establish a property ϕ for a system (module composition) $\mathcal{G}M$, it is sufficient to establish (if possible) ϕ for one of the components of $\mathcal{G}M$ (e.g. for $\mathcal{G}M_1$, the "decomposition" $\phi \equiv_{Def} \phi_1 \wedge \mathbf{true}$ justifies the argument with the according rule instantiation).

The main theorem which justifies rule R–comp makes extensive use of the next lemma.

Lemma 4.1 (Variable restriction) *Assume a set* $V \subseteq \mathcal{V}$ *of typed variables, a subset* $\mathcal{V}_1 \subseteq V$*, and an assertion language* $A\mathcal{L}$*. For* $E \in A\mathcal{L}$*, denote by* free(E) *the set of free variables in* E*. For a temporal logic formula* $\phi \in LTL_{A\mathcal{L}}$ *built over assertions* $E_1 \ldots E_k$*,* $k \geq 1$*, define* free$(\phi) =_{Def} \bigcup_{i=1}^{k}$ free(E_i).

Then

$$\forall \rho \in Val(V), \tilde{\rho} \equiv_{Def} \rho_{|V_1}, E \equiv_{Def} E^{Boolean} \in A\mathcal{L} :$$
$$free(E) \subseteq V_1 \implies [\![E]\!]\rho = \mathbf{true} \Leftrightarrow [\![E]\!]\tilde{\rho} = \mathbf{true} \quad , \quad (1)$$

and

$$\forall \sigma \in Comp(V), \tilde{\sigma} \equiv_{Def} \sigma_{|V_1}, \phi \in LTL_{A\mathcal{L}} :$$
$$free(\phi) \subseteq V_1 \implies \sigma \models \phi \Leftrightarrow \tilde{\sigma} \models \phi \quad . \qquad (2)$$

where $\sigma_{|V_1} =_{Def} (\rho_{i|V_1})_{i\geq 0}$ *if* $\sigma = (\rho_i)_{i\geq 0}$.

The next theorem is the justification for the rule R–comp.

Theorem 4.1 (Composition theorem) *Assume a set* V *of variables and an assertion language* $A\mathcal{L}$*, OTGS* $\mathcal{G}M_1$*,* $\mathcal{G}M_2$*, and formulas* ϕ_1*,* ϕ_2 *over* $V_i \equiv_{Def} V_{\mathcal{G}M_i}$ *(for* $i = 1, 2$*); let* $\mathcal{G}M$ *be a composition of* $\mathcal{G}M_1$ *and* $\mathcal{G}M_2$*,* $V \equiv_{Def} V_{\mathcal{G}M}$*. Then*

$$\mathcal{G}M_1 \models \phi_1 \textbf{ and } \mathcal{G}M_2 \models \phi_2 \implies \mathcal{G}M \models \phi_1 \wedge \phi_2 \quad .$$

Proof outline . Assume that $\mathcal{G}M_1 \models \phi_1$ **and** $\mathcal{G}M_2 \models \phi_2$ (*), i.e.

$$\forall \sigma \in L(\mathcal{G}M_1) \ : \ \sigma \models \phi_1 \text{ and } \forall \sigma \in L(\mathcal{G}M_2) \ : \ \sigma \models \phi_2$$

Show that

$$\forall \sigma \in L(\mathcal{G}M) \ : \ \sigma \models \phi_1 \wedge \phi_2 \qquad (**)$$

Assume that (**) does *not* hold, i.e.

$$\exists \sigma_0 \in L(\mathcal{G}M) \ . \ \sigma_0 \not\models \phi_1 \wedge \phi_2 \qquad (**\neg)$$

W.l.o.g we may assume that $\sigma_0 \not\models \phi_1$;

$\hookrightarrow$[$\sigma_0 \in L(\mathcal{G}M)$] $\sigma_0 \in L(TS_{\mathcal{G}M})$.

σ_0 is a computation over V, i.e. a sequence $(\rho_i)_{i\geq 0}$ of states over V. By definition, $\sigma_0 \in L(TS_{\mathcal{G}M})$ implies (with $TS =_{Def} TS_{\mathcal{G}M}$):

$\hookrightarrow$[Initiation] $[\![\Theta_{TS}]\!]\rho_0 = \mathbf{true}$ (***) and

$\hookrightarrow$[Consecution] σ_0 is "justified" by some transition sequence $\sigma\tau =_{Def} (\tau_i)_{i\geq 0}$ of transitions contained in $\mathcal{T} \equiv_{Def} \mathcal{T}_{\mathcal{G}M}$, i.e.

$$\forall i \geq 0 \ : \ \rho_{i+1} \in \tau_i(\rho_i) \qquad (****)$$

It remains to show that under assumption (**$\neg$) we may conclude that

$$\sigma_1 \equiv_{Def} \sigma_0{}_{|V_1} \in L(\mathcal{G}M_1) \qquad (*****)$$

Since $V_1 \subseteq V$ and $free(\phi_1) \subseteq V_1$, lemma 4.1, fact(2) applies; in particular $\hookrightarrow[\sigma_0 \not\models \phi_1]$ $\sigma_1 \not\models \phi_1$ which is together with (*****) a contradiction to premise (*); hence (**) follows from (*).

The rest of the proof of theorem 4.1 is given in the appendix.

4.5 Kripke–structures

We consider next a slightly simplified version of example 4.1.

Example 4.3 (Basic Req/Ack–handshake protocol) *We consider a "pure" version of the 4–phase asynchronous handshake protocol, assuming again type Bit with value domain* $\mathcal{D}^{Bit} =_{Def} \{$*'0', '1'*$\}$.

We define a TGS as follows:

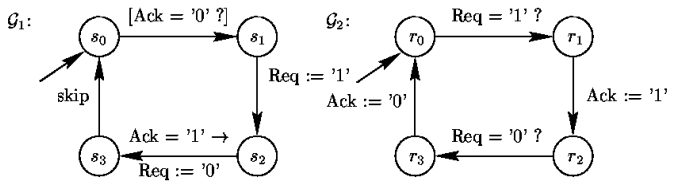
$$\begin{aligned} \mathcal{G}S^{basic}_{Req,Ack} \ : \quad & \textbf{program } \textit{Basic\_Req\_Ack\_Protocol} \\ & \textbf{out } \text{Req} : \text{Bit } \textbf{where } \text{Req} = \text{'0'} \\ & \textbf{out } \text{Ack} : \text{Bit } \textbf{where } \text{Ack} = \text{'0'} \\ & \mathcal{G}_1 \| \mathcal{G}_2 \end{aligned}$$

where $\mathcal{G}_1, \mathcal{G}_2$ *are defined in the diagrammatic presentation of the graphs shown in figure 4.3.*

In order to get a comprehensive overview over the possible computations of the program $\mathcal{G}S \equiv_{Def} \mathcal{G}S'_{Req,Ack}$, we introduce the notion of a *Kripke–structure*, which can be considered as a special case of a transition graph.

Definition 4.10 (Kripke–structure with fairness) *Assume a finite set* V *of variables and an assertion language* $A\mathcal{L}$*. A Kripke–structure (with fairness requirements)* $\mathcal{K}$ *(KS) is a structure*

Figure 4.3: TGS model of basic 4–phase handshake protocol.



$$(V, Locs, \mathcal{K}Edges, \ell_0, \theta_0, E\bar{\mathcal{J}}, E\mathcal{C}) \quad ,$$

where $(Locs, \mathcal{K}Edges, \ell_0, E\bar{\mathcal{J}}, E\mathcal{C})$ *is a transition graph (as defined in definition 4.3) and*

- $\mathcal{K}Edges$ *is a set of edges such that each edge is a triple*

  $$(\ell, \text{label}, \ell')$$

  where ℓ, ℓ' *are control locations and label is a constant assignment, which has the form*

  $$\bar{x} := \bar{c}$$

  where $\bar{x} \equiv [x_1, \ldots, x_k]$ *is a complete list of the variables of* V *in some fixed order (without doubles)* ($\{x_1, \ldots, x_k\} = V$) *and* $\bar{c} \equiv [c_1, \ldots, c_k]$ *is a list of constants, such that* x_i *and* c_i *have the same type* ($i = 1 \ldots k$).

- *All edges ending at the same location have the same label, i.e.*

  $$\begin{aligned} \forall e_1, e_2 \ : \quad & \\ & e_1 \equiv_{Def} \quad (\ell_1, label_1, \ell') \in \mathcal{K}Edges \wedge \\ & e_2 \equiv_{Def} \quad (\ell_2, label_2, \ell') \in \mathcal{K}Edges \\ & \qquad\qquad \rightarrow label_1 = label_2 \quad . \end{aligned}$$

- *For each location* $\ell \in Locs$ *there is an outgoing edge, i.e.*

  $$\forall \ell \exists \ell', label \ . \ (\ell, label, \ell') \in \mathcal{K}Edges$$

- *There is no edge which ends at the initial location* ℓ_0.
- *The initial valuation* θ_0 *is a mapping of the variables to some initial constants,*

$$\theta_0 \ : \ x \in V \mapsto c_{0,x} \quad ,$$

such that $c_{0,x}$ *has the same type as* x, $\forall x \in V$.

The diagrammatic representation of a Kripke–structure is similar to that of a transition graph, with one exception: The (common) label $\bar{x} := \bar{c}$ of any edge entering the (destination–)location ℓ' is denoted next to ℓ' in the assertional form $\bar{x} = \bar{c}$, or (if the vector $\bar{x}$ is clear from the context) simply in the form $\bar{c}$.

For the initial location, the initial valuation is denoted in the form $\bar{x} = \bar{c_0}$, where $\bar{c_0} \equiv_{Def} [c_{0,x_1}, \ldots, c_{0,x_k}]$ (or simply as $\bar{c_0}$).

An edge $e \in E\bar{\mathcal{J}}$, which is excepted from the requirement of justice, is drawn as a dashed arrow; an edge $e \in E\mathcal{C}$, which is subject to the requirement of compassion, is drawn as a solid arrow with a double head.

The semantics $L(\mathcal{K})$ of a Kripke–structure $\mathcal{K}$ over a set of variables V is a subset $L(\mathcal{K}) \subseteq Comp(V)$, which is defined via the semantics of an associated fair transition system $TS_{\mathcal{K}}$.

Definition 4.11 (Semantics of a Kripke–structure) *Assume a finite set* V *of variables, an assertion language* $A\mathcal{L}$ *and a Kripke–structure (KS)*

$$\mathcal{K} \ : \ (V, Locs, \mathcal{K}Edges, \ell_0, \theta_0, E\bar{\mathcal{J}}, E\mathcal{C}) \quad .$$

Then the semantics of $\mathcal{K}$, *denoted by* $L(\mathcal{K})$, *is defined to be the set of computations of the FTS*

$$TS_{\mathcal{K}} \equiv_{Def} (V', \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$$

i.e.: $L(\mathcal{K}) =_{Def} L(TS_{\mathcal{K}})$.

The components of $TS_{\mathcal{K}}$ *are defined as follows:*

- $V' =_{Def} V \dot{\cup} \{\pi_{\mathcal{K}}\}$, *where the fresh variable* $\pi_{\mathcal{K}} \notin V$ *represents the current control location in the KS* $\mathcal{K}$.
- $\Theta =_{Def} \mathbf{and}\!\!\!\!\bigwedge_{x \in V}(x = \theta_0(x))$,
- $\mathcal{T}$ *is the set of transitions defined by the following rule (R): For each labelled edge* $e \equiv_{Def} (\ell_1, x_1, \ldots, x_k := c_1, \ldots, c_k, \ell_2)$, $e \in Edges_i$ *and* $k \geq 0$, $\mathcal{T}$ *contains a transition* τ_e, *defined by*

$$\begin{array}{lll} \tau_e \quad =_{Def} (\rho & \mapsto & \emptyset, \quad \textit{if } [\![(\pi_{\mathcal{K}} = \ell_1)]\!]\rho = \textbf{false} \quad ; \\ & \mapsto & \{\rho' \in \mathit{Val}(V) \mid (\rho(\pi_{\mathcal{K}}) = \ell_2) \textit{ and } \rho'(x_i) = [\![c_i]\!]\rho, i = 1 \ldots k, \\ & & \quad \textit{and } \forall y \in V \backslash \{x_1, \ldots, x_k\} \,.\, \rho'(y) = \rho(y)\} \quad , \\ & & \quad \textit{if } [\![(\pi_{\mathcal{K}} = \ell_1)]\!]\rho = \textbf{true} \quad). \end{array}$$

The set $\mathcal{T}$ contains exactly the transitions defined by rule (R).

- $\mathcal{J} =_{Def} \{\tau_e \mid e \in \mathcal{K}Edges \backslash E\bar{\mathcal{J}}\}$ *is the set of transitions subject to the requirement of justice; and*
- $\mathcal{C} =_{Def} \{\tau_e \mid e \in E\mathcal{C}\}$ *is the set of transitions subject to the requirement of compassion.*

A Kripke–structure is a convenient representation to calculate the effects of the transitions of a TGS, while retaining the control–information of each transition graph in the TGS; the control–information is captured in according designated variables ranging over the respective domain of control locations.

Example 4.4 (Kripke–structure of the basic Req/Ack–handshake protocol) *We claim that the computations of the TGS $\mathcal{GS}'_{Req,Ack}$ are given by the semantics of the Kripke–structure*

$$\mathcal{K}' \;:\; (V, \mathit{Locs}, \mathcal{K}Edges, \ell_0, \theta_0, E\bar{\mathcal{J}}, E\mathcal{C}) \quad .$$

where $V = [Req, Ack, \pi_1, \pi_2]$ (π_i ranges over the control locations of the TG $\mathcal{G}_i$, for $i = 1, 2$), $\mathit{Locs} = \{\ell_0\} \cup \{\ell_{i,j} \mid 0 \leq i, j \leq 3\}$ and the other components are shown in the diagrammatic presentation of the Kripke–structure in figure 4.4.

Except for the initial location ℓ_0, all of the displayed locations $\ell_{i,j}$ have a loop–edge $(\ell_{i,j}, \_, \ell_{i,j}) \in \mathcal{K}Edges$, which is, however, shown explicitly only for the location $\ell_{0,0}$.

4.6 Summary

We have presented in this chapter the outline of a formal foundation of a verification environment, which has actually been implemented for verification of VHDL–based hardware designs and STATEMATE^{TM}–designs.

Figure 4.4: Kripke–structure for the basic Req/Ack–Protocol.

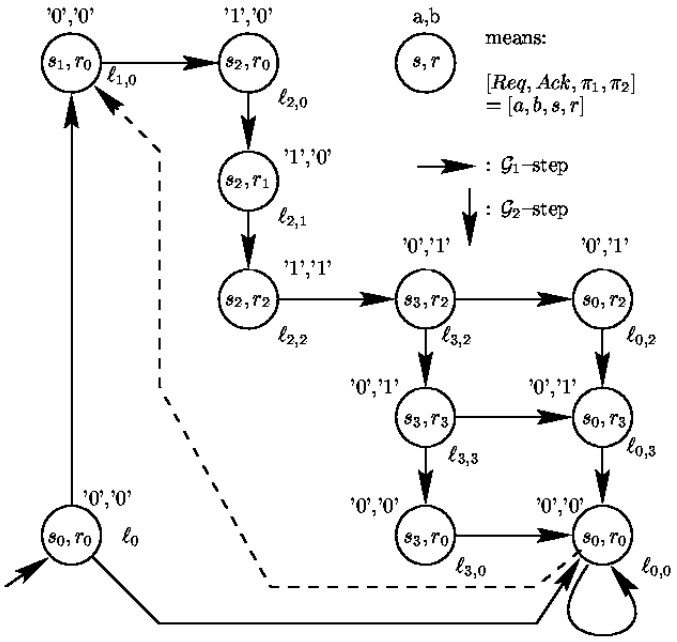
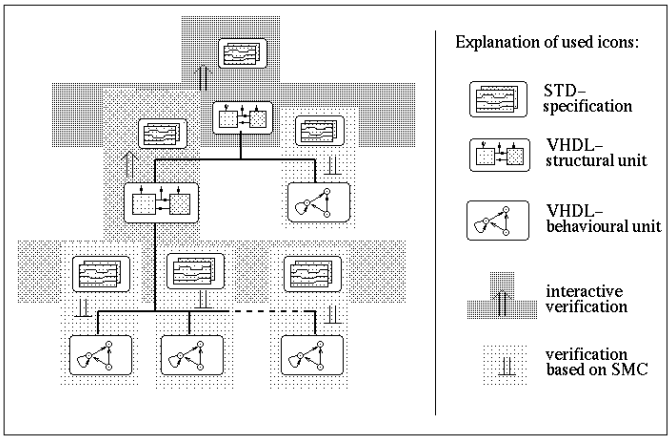


Figure 4.5: Divide–and–conquer strategy.



Consider for the following discussion figure 4.5 (taken from [30]).

It is seldom possible to verify properties of full ASICs or of full models of embedded systems of the typical size used e.g. in the area of automotive electronics *directly* using model–checking.

Thus, the following approach has been successfully applied to large designs:

- The property is split at the top–level composition into a conjunction of "local" properties, which are guaranteed by the sub–components.
- If the sub–components can be verified using the model–checking approach, then we are done. If not,
- split the specification of large sub–components again into sub–specifications of the processes (corresponding e.g. to specific functions of an embedded controller) of the large subcomponents.
- Verify the local properties of the sub–components using model–checking;
- Verify that the local specifications *imply* (in the sense of temporal logic) the global property.

In practice, we seldom encountered a case where more than a two–level decomposition was necessary.

Clearly, splitting a property logically into local properties, which together imply the global property, can become a hard task. Chances are good in those cases, where the sub–components have clean interfaces and cooperate in a well–defined way (e.g. use global shared variables only in a disciplined way).

This approach has also been extended (in a prototype implementation) to treat compositions of an arbitrary number of N identical sub–components, as reported in [30]. This approach relies on a theorem prover (e.g. the LAMBDA–prover) and on an axiomatization of the underlying assertion language (e.g. a selected subset of the language of VHDL–Boolean expressions).

Although powerful, the use of a theorem prover is as of today not accepted in an industrial environment. Therefore, logical derivations are performed in practice using a tableaux–based decision procedure called *tautology–checking*.

Chapter 5

Linear Symbolic Timing Diagrams

We introduce the formalism of Symbolic Timing Diagrams in two steps: First, we consider a syntactical subset of STD, called *Linear* Symbolic Timing Diagrams, which has an easily understandable semantics.

We will consider several aspects which are of interest w.r.t. the verification methodology, in particular

- Semantics in terms of Symbolic Automata and temporal logic
- Monotonicity properties, and
- Derivation rules.

Recall that our main goal is the design of a visual formalism for property specification. Why do we need a visual formalism? Temporal logic is already a concise and powerful formalism for the specification of reactive systems, and many tools have been built using temporal logic as input format for property specification (e.g. the SMV–system, [27]).

The main problem with a mathematical notation — e.g. the notation of temporal logic — is documentation. It is possible to write down a number of formulas, equipped with additional verbal comment, but large specifications constructed in that way become hard to understand.

A realistic document for a requirement specification may contain in the order of 100 distinct statements (this is a typical figure taken from the design of an embedded controller in the automotive area).

The hope of a visual formalism is to be at the same time rigorous and to a high degree self–documenting. Several developments of visual formalisms have been made in the past 10 years based on this motivation.

An example of a visual language which has gained widespread acceptance is the *Statecharts*–formalism invented by D. Harel [17]. This formalism is operational in nature and is particularly suited for graphical presentation, integrating notions of hierarchy and parallel execution.

5.1 Motivation

From the Kripke–structure $\mathcal{K}_{Req,Ack}$ shown in example 4.4 we can see, that all computations of the TGS $\mathcal{GS}^{basic}_{Req,Ack}$ result from the infinite repetition of a cyclic pattern $\ell_{0,0}, \ldots \ell_{0,0}, \ell_{1,0}, \ldots, \ell_{0,0}$ (for the very beginning of each computation, ℓ_0 instead of $\ell_{0,0}$), representing the frame of a bus read– or write–cycle after some idle period (spent at location $\ell_{0,0}$). Since the edge from $\ell_{0,0}$ to $\ell_{1,0}$ is excluded from the requirement of justice, eventually there may be no further cycle initiation.

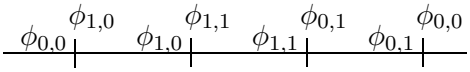
For each cycle, there are actually three possible computation paths in order to move from $\ell_{1,0}$ to $\ell_{0,0}$ (via either of the locations $\ell_{0,2}$, or $\ell_{3,3}$ followed by $\ell_{0,3}$, or $\ell_{3,0}$), dependent on the "relative speed" of the atomic actions in $\mathcal{G}_1$ and $\mathcal{G}_2$.

These differences are, however, not observable at the interface variables (Req, Ack), whose valuations in any case follow the pattern

$$\begin{aligned} Req: &\quad \text{'0'}\ldots\text{'0''1'}\ldots\text{'1'}\ldots\text{'0'}\ldots\text{'0'} \\ Ack: &\quad \text{'0'}\ldots\text{'0''0'}\ldots\text{'1'}\ldots\text{'1'}\ldots\text{'0'} \end{aligned}$$

"Linear" parts of a behaviour (e.g. of part of a protocol) are tedious to express in temporal logic, while a graphical representation is natural. We note that the sequence shown above consists of an alternating sequence of "stuttering" steps — which are steps where no value changes occur — and steps, where a value change has occurred.

Consider the following alternative way to present this sequence:

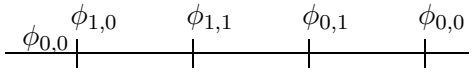


where

$$\begin{aligned}
\phi_{0,0} = &\quad \langle\!| Req = \text{'0'} \text{ and } Ack = \text{'0'} |\!\rangle \\
\phi_{1,0} = &\quad \langle\!| Req = \text{'1'} \text{ and } Ack = \text{'0'} |\!\rangle \\
\phi_{1,1} = &\quad \langle\!| Req = \text{'1'} \text{ and } Ack = \text{'1'} |\!\rangle \\
\phi_{0,1} = &\quad \langle\!| Req = \text{'0'} \text{ and } Ack = \text{'1'} |\!\rangle
\end{aligned}$$

We may further simplify the graphical notation using the following convention: If the stuttering steps following a value change are characterized by the same formula as the one used to specify the new values reached by the last change, then this formula can be omitted.

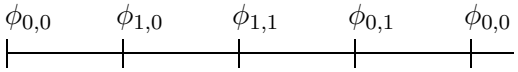
Using this convention, we present the Req, Ack–sequence as follows:



More precisely, we might want to express that whenever the interface variables Req, Ack have the value '0', '0' , then the following sequence of stuttering steps and value–changes must be as specified above.

We will use the following notation to express the "whenever" part of the statement:

Using the convention not to repeat identical formulas, we may now combine the "whenever"–part and the "sequence"–part by juxtaposition as follows:



which is read in the following way: Whenever the value of Req, Ack is '0', '0', then the values in subsequent steps must conform to the displayed sequence pattern.

It remains to clarify what happens at the *end* of the sequence specification. We adopt the convention, that the pattern is said to be *completely matched* after the last formula (here formula $\phi_{0,0}$) has been matched by the actual behaviour. Then, *no further restriction is expressed.*

Note, however, that the semantics of the "whenever"–clause becomes active again at the point of satisfaction of the final formula $\phi_{0,0}$, and insists on the

repeated satisfaction of the sequence.

We can make the requirement specification complete by stating what happens after system initialization.

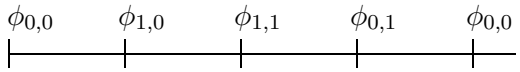
This is expressed in the form:

which says that in step 0 the system interface variables must satisfy $\phi_{0,0}$.

Thus, a complete specification of the basic Req/Ack–protocol is displayed as follows:

Initialization:

Invariance:



This is essentially the basis of the semantics definition of Symbolic Timing Diagrams, presented here for the special case that the diagrams consist of one symbolic waveform only.

We can state the following facts about STD so far:

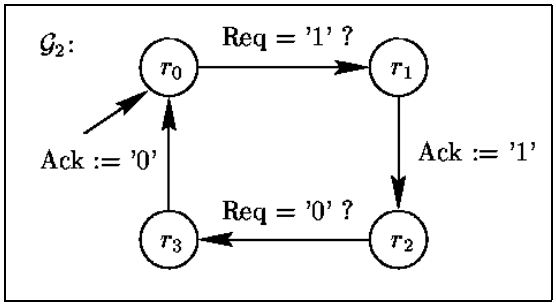
- A STD–*specification* consists of a set of diagrams (which are interpreted conjunctively), and
- STD–*diagrams* always have one of two possible *activation modes*: Either Initial or Invariant.

Now, reconsider the *Slave*–module of the *distributed* version of the Req/Ack–protocol introduced in example 4.2.

The slave–module was defined as

$$\begin{aligned} \mathcal{G}M_{Ack} : \quad & \textbf{module} \ \text{Slave} \\ & \textbf{out} \ \text{Ack : Bit} \ \textbf{where} \ \text{Ack} = \text{'0'} \\ & \textbf{external} \ \text{Req: Bit} \\ & \mathcal{G}_2 \end{aligned}$$

Figure 5.1: Implementation of Slave–model of the basic 4–phase handshake protocol.

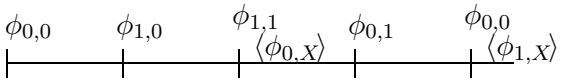


where $\mathcal{G}_2$ is defined as shown in figure 5.1.

We would like to construct a "local" specification of the Slave–module with respect to the interface variables Req, Ack . The important point to consider now is that the variable Req is *external*, which means that it can be set by the environment of the module to any value in each step.

Of course, we *know* that we will finally compose the Slave–module with a "matching" Master–module in a way, that a proper 4–phase handshake protocol is implemented by the composition.

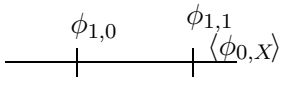
This brings us to the following suggestion of a local specification of the sequence part of the Req/Ack–protocol:



where

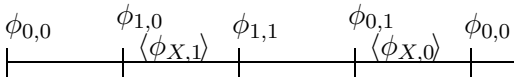
$$\begin{aligned}\phi_{0,X} = &\quad \langle\!|Req = \text{'0'} \text{ and } \mathbf{true}|\!\rangle \sim \langle\!|Req = \text{'0'}|\!\rangle \\ \phi_{1,X} = &\quad \langle\!|Req = \text{'1'} \text{ and } \mathbf{true}|\!\rangle \sim \langle\!|Req = \text{'1'}|\!\rangle\end{aligned}$$

The formula in angle brackets is called an *exit–condition*. Consider the following fragment of the sequence specification:



The exit condition relaxes the requirement, that any change after a sequence of states which satisfy formula $\phi_{1,0}$ must lead to new values which satisfy formula $\phi_{1,1}$. In addition, the change is now allowed to lead to new values which satisfy the exit–condition $\phi_{0,X}$. This is a legal (!) behaviour with respect to the specification. It causes an "exit" from the matching process, which means that the diagram terminates its restricting effect. An exit–condition is a critical point with respect to a correct system specification. It means that the specification depends on a proper complementation by other diagrams (contributed from the specification of other modules).

In this case, a proper complementing specification of the Req/Ack–sequence contributed from the master–module is as follows:



where

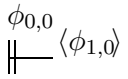
$$\phi_{X,0} = \quad \langle\!|\mathbf{true} \text{ and } Ack = \text{'0'}|\!\rangle \sim \langle\!|Ack = \text{'0'}|\!\rangle$$
$$\phi_{X,1} = \quad \langle\!|\mathbf{true} \text{ and } Ack = \text{'1'}|\!\rangle \sim \langle\!|Ack = \text{'1'}|\!\rangle$$

The conjunctive effect of the local sequence specification of the Slave and the local sequence specification of the Master implies the sequence specification of the *Req/Ack*–protocol, because at all points, where an exit might occur, this is prohibited by the other diagram.

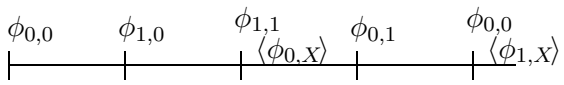
The initial specification has to be adapted in a similar way for the local specification of the Slave module, because the initial values of external variables is determined by the environment. Again, adding an exit–condition to an initial specification, relaxes the requirement on the initial condition.

Thus, the following local specification is obtained for the Slave–module:

Initialization:



Invariance:

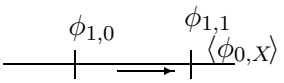


Let us take another look at the implementation of the Slave module, shown

in figure 5.1. At (the initial) location r_0, the module waits for an assertion of the *Req* signal (*Req* = '1'). If this happens, the transition to location r_1 is enabled.

If *Req remains asserted* (*Req* = '1'), then the requirement of justice demands that the transition will eventually be taken, transferring control to location r_1. At this point, the transition to r_2 is (unconditionally) enabled. Again, the requirement of justice demands that the transition will eventually be taken, transferring control to location r_2, and asserting *Ack* (*Ack* = '1').

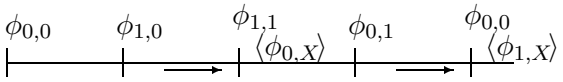
This can be expressed graphically in the following way:



The arrow under the timeline after the point where formula $\phi_{1,0}$ holds means that something must eventually (in the sense of temporal logic) happen with respect to the values of *Req*, *Ack*.

There are two possibilities: Either, *Ack* is asserted (which is the expected behaviour), of *Req* is de–asserted (which is an unexpected behaviour from the environment). A proper implementation of the Master–module guarantees that de–assertion of *Req* cannot happen at this point, hence the expected reaction (assertion of *Ack*) will eventually occur.

The final sequence specification of the Slave–module is as follows:



5.2 Syntax of LSTD

We will now introduce the formalism of Linear Symbolic Timing Diagrams (LSTD) formally. The idea is that a sequence specification, called an LSTD–*body*, can be considered to consist of a sequence of LSTD–*phases*, which is terminated by a (final) LSTD–phase.

Definition 5.1 (LSTD–body) *Assume an assertion language $A\mathcal{L}$, and a set $\mathcal{V}$ of variables. Then the language of Linear Symbolic Timing Diagram bodies (LSTD–body), denoted by Δ) is defined by the following grammar with the production sets (1)–(4):*

$$\Delta \longrightarrow \Delta E \qquad (1)$$

$$\mid \quad \Delta E\ \Delta_1 \qquad (2)$$

$$\Delta E \longrightarrow \phi_1 \mid \langle \phi_3 \rangle^{\phi_2} \qquad (3)$$

$$\mid \quad \phi_1 \mid \langle \phi_3 \rangle^{\phi_2} \qquad (4)$$

where $\phi_1, \phi_2, \phi_3 \in Bool_{A\mathcal{L}}$ are Boolean formulas over $\mathcal{V}$. The meaning of these productions is as follows: Production (1) and (2) define a LSTD–body to be a non–empty sequence of LSTD–phases, (also called phase, for short) where a LSTD–phase is denoted in one of the graphical forms shown in production (3) respectively (4). Note that a sequence of phases is formed by graphical juxtaposition such that the impression of a single graphical object is obtained, as e.g. in

$$\phi_1 \mid \langle \phi_3 \rangle \phi_4 \mid \langle \phi_6 \rangle^{\phi_5}{}^{\phi_2} \ldots$$

An LSTD–body is not yet a diagram: It must be closed on the left side by a so–called *activation–specification*.

The activation specification is an essential part of the semantics of LSTD (and STD); it decides whether a sequence specification is required either to hold initially, or whenever a specified state–formula holds.

Definition 5.2 (LSTD Diagram) *Assume an assertion language $A\mathcal{L}$, and a set V of variables. Then the language of Linear Symbolic Timing Diagrams (LSTD, denoted by $\alpha\Delta$) is defined by the following grammar with the production sets (1)–(3):*

$$\alpha\Delta \longrightarrow \alpha E\ \Delta \qquad (1)$$

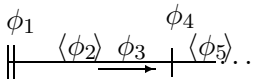
$$\alpha E \longrightarrow \vdash^{\phi_1} \qquad (2)$$

$$\mid \quad \vdash\!\!\vdash^{\phi_1} \langle \phi_2 \rangle \qquad (3)$$

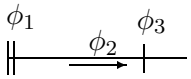
where $\phi_1, \phi_2 \in Bool_{A\mathcal{L}}$ are Boolean formulas over V, and Δ is a LSTD–body as defined by definition 5.1. The productions (2) and (3) define activation

specifications: The first type, defined by production (2), means that whenever a system state occurs which satisfies ϕ_1, then the further computation must conform to the semantics of the LSTD–body Δ; the second type, defined by production (3), means that the initial system state must either *satisfy ϕ_1 and then the further computation must conform to the semantics of the LSTD–body Δ,* or *satisfy ϕ_2(the so–called initial–exit–specification).*

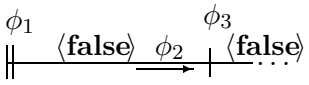
Note that a diagram is formed by graphical juxtaposition of an activation specification and an LSTD–body such that the impression of a single graphical object is obtained, as e.g. in



We adopt for LSTD–bodies and for activation specifications the convention, that an (initial–)exit–specification can be omitted, in which case it is assumed to be **false** by default. E.g.,



is equivalent to



5.3 Semantics of LSTD

We will define the semantics of the language LSTD by a translation of LSTD–bodies into Symbolic Automata and an explicit definition of the semantics of activation specifications, which together can be turned into a characterizing temporal logic formula. This opens the way for using LSTD in a formal verification framework based on model–checking and tautology–checking.

In preparation of the next definition, we need to make a small extension to the notion of a POSA. We say that a POSA $\mathcal{A}$ has *a top state*, denoted by

$$\ell^{top}(\mathcal{A}) \qquad \text{(unique top element of } \mathcal{A})$$

if the partial order $\preceq_{\mathcal{A}}$ (cf. definition 3.7)has a *unique maximal element*. For a POSA A, $\ell^{top}(\mathcal{A})$ is *undefined*, is $\mathcal{A}$ has no unique maximal element.

The next definition will provide a SA of a LSTD–body, which is by construction (in all cases) a POSA with a top state. Furthermore, the construction ensures that the SA of a LSTD–body has a unique initial state.

Definition 5.3 (SA of LSTD–body) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and a LSTD–body Δ over V. Then we define the associated SA $\mathcal{A}_\Delta$ to be the SA*

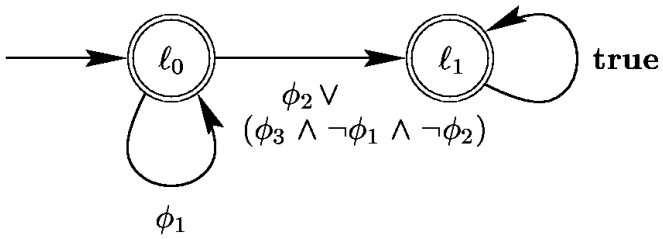
$$\mathcal{A}_\Delta \equiv_{Def} (V, Locs, Edges, \{\ell_0\}, F)$$

by induction on the structure of Δ:

- *Production sequence: $1 \rightarrow 3$: Assume that $\Delta \equiv_{Def} \Delta E$ is a single LSTD phase; assume further that*

$$\Delta E \equiv_{Def} \phi_1 \; \overset{\phi_2}{\vdash} \; \langle \phi_3 \rangle \; .$$

Then the components of $\mathcal{A}_\Delta$ are defined as follows: $Locs =_{Def} \{\ell_0, \ell_1\}$, $Edges =_{Def} \{(\ell_0, \phi_1, \ell_0), (\ell_0, \phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2), \ell_1), (\ell_1, \mathbf{true}, \ell_1)\}$ and $F =_{Def} \{\ell_0, \ell_1\}$; $\mathcal{A}_\Delta$ is graphically depicted as



- *Production sequence: 1 → 4: In the case that*

$$\Delta E \equiv_{Def} \stackrel{\phi_2}{\phi_1 \Longrightarrow \langle\phi_3\rangle}$$

all components of $\mathcal{A}_\Delta$ are defined as in the previous case ($\Delta E \equiv_{Def} \stackrel{\phi_2}{\phi_1 \Longrightarrow \langle\phi_3\rangle}$), with the exception that the first location ℓ_0 of $\mathcal{A}_\Delta$ is not an acceptance state, i.e. $F =_{Def} \{\ell_1\}$.

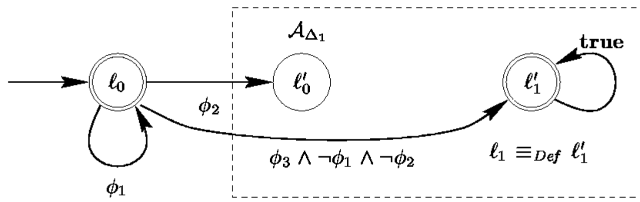
- *Production sequence: 2 → 3 → ...: This case is the inductive step of the definition of LSTD–bodies. We consider a LSTD–body of the form $\Delta \equiv_{Def} \Delta E\ \Delta_1$, and assume that*

$$\mathcal{A}' \equiv_{Def} (V, Locs', Edges', \{\ell'_0\}, F')$$

is the SA constructed for Δ_1, $\mathcal{A}' \equiv_{Def} \mathcal{A}_{\Delta_1}$; let $\ell'_1 \equiv_{Def} \ell^{top}(\mathcal{A}')$. First, assume that

$$\Delta E \equiv_{Def} \stackrel{\phi_2}{\phi_1 \Longrightarrow \langle\phi_3\rangle}\ .$$

Then the components of $\mathcal{A}_\Delta$ are defined as follows: $Locs =_{Def} \{\ell_0\} \cup Locs'$ (where ℓ_0 is a new location, $\ell_0 \notin Locs'$); with $\ell_1 \equiv_{Def} \ell'_1 = \ell^{top}(\mathcal{A})$, $Edges =_{Def} \{(\ell_0, \phi_1, \ell_0), (\ell_0, \phi_2, \ell'_0), (\ell_0, (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2), \ell_1)\} \cup Edges'$ and $F =_{Def} \{\ell_0\} \cup F'$; $\mathcal{A}_\Delta$ is graphically depicted as



- *Production sequence: 2 → 4 → …: In the case that*

$$\Delta E \equiv_{Def} \underset{\longrightarrow}{\phi_1} \overset{\phi_2}{|} \langle\phi_3\rangle$$

all components of $\mathcal{A}_\Delta$ are defined as in the previous case ($\Delta E \equiv_{Def}$ $\phi_1 \overset{\phi_2}{|} \langle\phi_3\rangle$), except that the first location ℓ_0 is not an acceptance state, i.e. $F =_{Def} F'$.

The definition of the semantics of LSTD–diagrams is a combination of a declarative definition (of the kind used for the semantics of temporal logic) and the semantics defined for Symbolic Automata.

Definition 5.4 (LSTD–diagram semantics) *Assume an assertion language $A\mathcal{L}$, a set $\mathcal{V}$ of variables, and a LSTD–diagram $\alpha\Delta$ over V. Then the semantics of $\alpha\Delta$, denoted by $L(\alpha\Delta)$, is defined as follows:*

- *case $\alpha\Delta \equiv_{Def} \overset{\phi_1}{\vdash} \Delta$: In this case,*

$$L(\alpha\Delta) =_{Def} \{\sigma \in Comp(V) \mid \forall k \geq 0 \ : \ \sigma(k) \models \phi_1 \rightarrow \sigma^{(k+1)} \in L(\Delta)\}$$

- *case $\alpha\Delta \equiv_{Def} \overset{\phi_1}{\Vdash} \langle\phi_2\rangle \, \Delta$: In this case,*

$$L(\alpha\Delta) =_{Def} \{\sigma \in Comp(V) \mid \sigma(0) \models \phi_2 \text{ or } \sigma(0) \models \phi_1 \wedge \sigma^{(1)} \in L(\Delta)\}$$

where $L(\Delta) \equiv_{Def} L(\mathcal{A}_\Delta)$ *is the semantics of the SA* $\mathcal{A}$ *obtained from* Δ *according to definition 5.3.*

We have shown already in chapter 2 that a STD–specification (and in particular an LSTD–specification) consists in general or a set of diagrams; the definition of the structure and the semantics of an LSTD–specification is given in the next definition.

Definition 5.5 (LSTD–specification) *Assume an assertion language* $A\mathcal{L}$*, and a set* V *of variables. A LSTD–specification (LSTD–Spec)* ΔS *is a finite set of LSTD–diagrams over* V*, i.e. for some* $k \geq 1$

$$\Delta S = \{\alpha\Delta_i \mid i = 1 \ldots k\} \quad .$$

The semantics of a LSTD–Spec ΔS*, denoted by* $L(\Delta S)$*, is defined to be the intersection of the semantics of the diagrams contained in* ΔS*, i.e.*

$$L(\Delta S) =_{Def} \bigcap_{i=1\ldots k} L(\alpha\Delta_i) \quad .$$

5.3.1 Translation from LSTD–diagrams to temporal logic

We know by theorem 3.3 that the semantics of a partially ordered Symbolic Automaton (POSA) can be characterized by a temporal logic formula, and how the formula is constructed from a given POSA.

First, we claim that the Symbolic Automaton constructed from LSTD–bodies according to definition 5.3 is a POSA. Then, we apply the construction given in theorem 3.3 to define a temporal logic formula characterizing the semantics of LSTD–bodies.

Lemma 5.1 (Structure of SA constructed from LSTD–body) *Assume an assertion language* $A\mathcal{L}$*, a set* V *of variables, and a LSTD–body* Δ *over* V*. Then the associated SA* $\mathcal{A}_\Delta$ *defined by the construction in definition 5.3 is a POSA.*

The proof follows directly from the construction given in definition 5.3: For the basic steps (Production sequence $1 \rightarrow 3$ and $1 \rightarrow 4$), the constructed automaton is clearly a POSA. For the inductive definition steps (Production

sequence $2 \rightarrow 3 \rightarrow \ldots$ and $2 \rightarrow 4 \rightarrow \ldots$), the added new first element has only (a loop and) outgoing transitions and therefore extends the partial order of the transition relation of the automaton assumed in the hypothesis of the inductive construction steps.

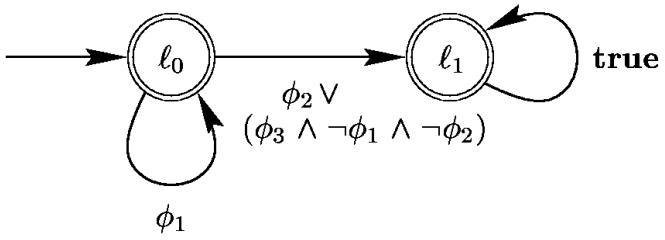
Therefore, we can apply theorem 3.3 to define the temporal logic formula characterizing the semantics an LSTD–body Δ .

Definition 5.6 (Formula characterizing the semantics of a LSTD–body) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and a LSTD–body Δ over V. Then we define the associated temporal logic formula ϕ_Δ by induction on the structure of Δ:*

- *Production sequence: $1 \rightarrow 3$: Assume that $\Delta \equiv_{Def} \Delta E$ is a single LSTD phase; assume further that*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\dashv\vdash}} \langle\phi_3\rangle \ .$$

Then by definition 5.3 , $\mathcal{A}_\Delta$ is defined as



Hence, according to theorem 3.3 ,

$$\begin{aligned} \phi_\Delta \quad =_{Def} \quad & \phi_1 \textbf{ unless} \\ & ((\phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)) \\ & \wedge \textbf{next} (\textbf{true unless false})) \\ \sim \quad & \phi_1 \textbf{ unless} \\ & (\phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)) \quad . \end{aligned}$$

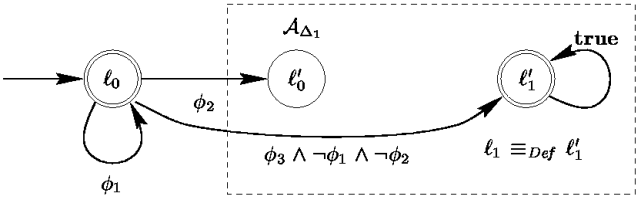
- *Production sequence: 1 → 4: Assume that* $\Delta \equiv_{Def} \Delta E$ *is a single LSTD phase, where*

$$\Delta E \equiv_{Def} \stackrel{\phi_2}{\stackrel{\phi_1}{\Longrightarrow}\!\!\mid}\langle\phi_3\rangle \ .$$

This case is similar to the preceding case (Production sequence: 1 → 3), except that for the associated POSA $\mathcal{A}_\Delta$, ℓ_0 *is not an accepting state. Hence, according to theorem 3.3 ,*

$$\begin{array}{rcl} \phi_\Delta & =_{Def} & \phi_1 \ \mathbf{until} \\ & & ((\phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)) \\ & & \wedge \mathbf{next}\ (\mathbf{true}\ \mathbf{unless}\ \mathbf{false})) \\ & \sim & \phi_1 \ \mathbf{until} \\ & & (\phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)) \quad . \end{array}$$

- *Production sequence: 2 → 3 → ...: This case is the inductive step of the definition of LSTD–bodies. We consider a LSTD–body of the form* $\Delta \equiv_{Def} \Delta E\ \Delta_1$, *and assume that* ϕ_{Δ_1} *is the formula constructed for* Δ_1. *By definition 5.3 ,* $\mathcal{A}_\Delta$ *is defined as*



Assume that

$$\Delta E \equiv_{Def} \stackrel{\phi_2}{\stackrel{\phi_1}{\Longrightarrow}\!\!\mid}\langle\phi_3\rangle \ .$$

Then according to theorem 3.3 ,

$$
\begin{array}{lcl}
\phi_\Delta & =_{Def} & \phi_1 \textbf{ unless} \\
 & & ((\phi_2 \wedge \textbf{next } \phi_{\ell'_0}) \\
 & & \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) \\
 & & \wedge \textbf{next } (\textbf{true unless false})) \\
[\phi_{\ell'_0} = \phi_{\mathcal{A}_{\Delta_1}} = \phi_{\Delta_1}] & \sim & \phi_1 \textbf{ unless} \\
 & & ((\phi_2 \wedge \textbf{next } \phi_{\Delta_1}) \\
 & & \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2))
\end{array}
$$

- *Production sequence: 2 → 4 → ...: We consider a LSTD–body of the form* $\Delta \equiv_{Def} \Delta E\ \Delta_1$, *and assume that*

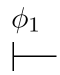
$$\Delta E \equiv_{Def} \overset{\phi_1}{\longrightarrow} \overset{\phi_2}{|} \langle\phi_3\rangle .$$

  Then $\mathcal{A}_\Delta$ *is defined as in the previous case (Production sequence: 2 → 3 → ...) except that* ℓ_0 *is not an accepting state. Hence, according to theorem 3.3 ,*

$$
\begin{array}{lcl}
\phi_\Delta & =_{Def} & \phi_1 \textbf{ until} \\
 & & ((\phi_2 \wedge \textbf{next } \phi_{\ell'_0}) \\
 & & \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) \\
 & & \wedge \textbf{next } (\textbf{true unless false})) \\
[\phi_{\ell'_0} = \phi_{\mathcal{A}_{\Delta_1}} = \phi_{\Delta_1}] & \sim & \phi_1 \textbf{ until} \\
 & & ((\phi_2 \wedge \textbf{next } \phi_{\Delta_1}) \\
 & & \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2))
\end{array}
$$

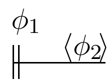
The next step is to determine the characterizing temporal logic formula for LSTD–diagrams.

Lemma 5.2 (Characterizing formula for LSTD–diagrams) *Assume an assertion language* $A\mathcal{L}$, *a set* V *of variables, and a LSTD–diagram* $\alpha\Delta$ *over* V. *Let formula* $\phi_{\alpha\Delta}$ *be defined by:*

- *case* $\alpha\Delta \equiv_{Def}$ 

Δ:

$$\phi_{\alpha\Delta} =_{Def} \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,\phi_\Delta) \quad ;$$

- *case* $\alpha\Delta \equiv_{Def}$ 

Δ:

$$\phi_{\alpha\Delta} =_{Def} \phi_2 \vee (\phi_1 \wedge \textbf{next}\,\phi_\Delta) \quad .$$

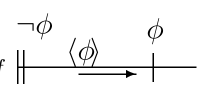
is a characterizing formula for the semantics of $\alpha\Delta$ *(as defined by definition 5.4), i.e.*

$$L(\alpha\Delta) = L(\phi_{\alpha\Delta})$$

The proof of lemma 5.2 follows immediately from the definition of the (temporal) logic operators used in the construction of formula $\phi_{\alpha\Delta}$.

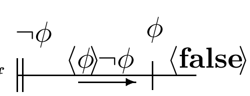
We give two examples to illustrate the definition of the construction of $\phi_{\alpha\Delta}$.

Example 5.1 (LSTD–diagram $\alpha\Delta_{\textbf{eventually}\ \phi}$) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and a LSTD–diagram $\alpha\Delta_{\textbf{eventually}\ \phi}$ over V, defined by*

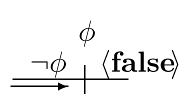
$\alpha\Delta_{\textbf{eventually}\ \phi} =_{Def}$ 

.

Applying the default rules for LSTD–diagrams about omission of exit–specification, this is equivalent to:

$\alpha\Delta_{\textbf{eventually}\ \phi} =_{Def}$ 

Define the LSTD–phase of $\alpha\Delta_{\textbf{eventually}\ \phi}$ *by*

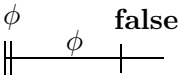
$\Delta_1 =_{Def}$ 

From lemma 5.2 we get:

$$\begin{aligned}
\phi_{\alpha\Delta} \quad &=_{Def} \quad \phi \vee (\neg\phi \wedge \mathbf{next}\, \phi_{\Delta_1}) \\
&\sim \quad \phi \vee (\neg\phi \wedge \mathbf{next}\, (\neg\phi\, \mathbf{until}\, \phi)) \\
&\sim \quad \phi \vee (\neg\phi \wedge \mathbf{next}\, (\mathbf{eventually}\, \phi)) \\
&\sim \quad \phi \vee \mathbf{next}\, (\mathbf{eventually}\, \phi) \\
&\sim \quad \mathbf{eventually}\, \phi \quad .
\end{aligned}$$

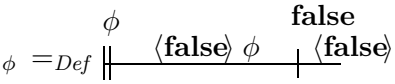
Example 5.2 (LSTD–diagram $\alpha\Delta_{\mathbf{always}\ \phi}$) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and a LSTD–diagram $\alpha\Delta_{\mathbf{always}\ \phi}$ over V, defined by*

$\alpha\Delta_{\mathbf{always}\ \phi} =_{Def}$



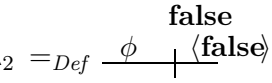
Applying the default rules for LSTD–diagrams about omission of exit–specification, this is equivalent to:

$\alpha\Delta_{\mathbf{always}\ \phi} =_{Def}$



Define the LSTD–phase of $\alpha\Delta_{\mathbf{always}\ \phi}$ by

$\Delta_2 =_{Def}$



From lemma 5.2 we get:

$$\begin{aligned}
\phi_{\alpha\Delta} \quad &=_{Def} \quad (\phi \wedge \mathbf{next}\, \phi_{\Delta_2}) \\
&\sim \quad (\phi \wedge \mathbf{next}\, (\phi\, \mathbf{unless}\, \mathbf{false})) \\
&\sim \quad (\phi \wedge \mathbf{next}\, (\mathbf{always}\, \phi)) \\
&\sim \quad \mathbf{always}\, \phi \quad .
\end{aligned}$$

Finally, the characterizing formula of an LSTD–specification is given by the next lemma (the proof is trival).

Lemma 5.3 (Characterizing formula of LSTD–specification) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–specification (LSTD–Spec) ΔS , where for some $k \geq 1$*

$$\Delta S \equiv_{Def} \{\alpha\Delta_i \mid i = 1 \ldots k\} \quad .$$

Then the formula

$$\phi_{\Delta SPEC} =_{Def} \bigwedge_{i=1\ldots k} \phi_{\alpha\Delta_i} \quad .$$

is a characterizing formula for the semantics of ΔS (as defined by definition 5.5), i.e.

$$L(\Delta S) = L(\phi_{\Delta SPEC})$$

This establishes an essential property of the conception of LSTD: An LSTD–specification can be translated into a characterizing temporal logic formula, and can therefore be used in a verification environment based on temporal logic model–checking and tautology–checking.

5.3.2 Translation from deterministic POSA to LSTD

We reconsider the concept of deterministic POSA introduced in chapter 3.

From theorem 3.5 we know, that a deterministic POSA $\mathcal{A}$ can be characterized by a formula $\xi_{\mathcal{A}} \in LINLTL_V$.

In this section, we will use the construction of the proof of theorem 3.5 again and show that a set of LSTD–bodies can be derived which characterizes the semantics of $\mathcal{A}$.

Theorem 5.1 (Translation from deterministic POSA to LSTD) *Assume an assertion language $A\mathcal{L}$, and let $\mathcal{A}$ be a deterministic POSA over some set V of variables. Then there exists a set of LSTD–bodies $\{\Delta_1, \ldots, \Delta_k\}$ such that*

$$L(\mathcal{A}) = \bigcap_{i=1\ldots k} L(\Delta_i) \quad . \qquad (*)$$

Proof of theorem 5.1 – Construction. The construction of the set $\{\Delta_1, \ldots, \Delta_k\}$ corresponds to the definition of the formulas ϕ_ℓ in theorem 3.3:

$$\phi_\ell =_{Def} \phi_{\ell,\ell} \quad \mathcal{U} \quad (\bigvee_{\ell \neq \ell' \,:\, \ell \to \ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\, \phi_{\ell'})$$

Note that $\mathcal{A}$ is deterministic and has therefore a unique initial location ℓ_0; hence $\phi_{\mathcal{A}} = \phi_{\ell_0}$.

We show that for each formula ϕ_ℓ, there is an equivalent set of LSTD–bodies

$$\Delta S_\ell \equiv_{Def} \{\Delta_1^\ell, \ldots, \Delta_{k(\ell)}^\ell\}$$

such that $L(\phi_\ell) = \bigcap_{i=1\ldots k(\ell)} L(\Delta_i^\ell)$.

The proof is by induction, using the same monotonic sequence $(T_i)_{i\geq 0}$ of sets as defined in the proof of theorem 3.3 .

The proof is based on the fact that set ΔS_ℓ can be presented as

$$\Delta S_\ell \equiv_{Def} \bigcup_{\ell' \neq \ell, \ell \to \ell'} \Delta S_{\ell,\ell'}$$

where

$$\Delta S_{\ell,\ell'} \equiv_{Def} \{ \underset{\rightsquigarrow}{\phi_{\ell,\ell}} \overset{\phi_{\ell,\ell'}}{\mid} \langle \phi_{\ell,\ell'} \rangle \; \Delta_1 \mid \Delta_1 \in \Delta S_{\ell'} \} \quad .$$

The notation $\underset{\rightsquigarrow}{\phi_{\ell,\ell}} \overset{\phi_2}{\mid} \langle \phi_3 \rangle$ means:

$$\underset{\rightsquigarrow}{\phi_{\ell,\ell}} \overset{\phi_2}{\mid} \langle \phi_3 \rangle \quad \text{if } \ell \in F \;;$$

$$\underset{\longrightarrow}{\phi_{\ell,\ell}} \overset{\phi_2}{\mid} \langle \phi_3 \rangle \quad \text{if } \ell \notin F \quad .$$

where F is the set of acceptance states of $\mathcal{A}$.

The notation $\bar{\phi}_{\ell,\ell'}$ is defined as in theorem 3.3 :

$$\bar{\phi}_{\ell,\ell'} \equiv_{Def} \bigvee_{\ell'' \neq \ell', \ell'' \neq \ell, \ell \to \ell''} \phi_{\ell,\ell''} \quad .$$

case $\ell \in T_0$ (set of final locations). For elements $\ell \in T_0$, no successor (different from ℓ) exists.

Define

$$\Delta S_\ell \equiv_{Def} \{\Delta_1^\ell\}$$

where

$$\Delta_1^\ell \equiv_{Def} \underset{\rightsquigarrow}{\phi_{\ell,\ell}} \mid \overset{\mathbf{false}}{\langle \mathbf{false} \rangle}.$$

Then

$$L(\Delta_1^\ell) = L(\phi_{\ell,\ell}\, \mathcal{U}\, ((\mathbf{false} \wedge \mathbf{next}\, \mathbf{true}) \vee \mathbf{false})) = L(\phi_\ell) \quad .$$

case $\ell \in T_{k+1} \backslash T_k$ **, some** $k \geq 0$ **.** We assume that for each $\ell' \in T_k$, there exists a set $\Delta S_{\ell'} \equiv_{Def} \Delta S'$ such that $L(\Delta S') = L(\phi_{\ell'})$.

The semantics of set $\Delta S_{\ell,\ell'}$ is characterized by

$$L(\Delta S_{\ell,\ell'}) \equiv_{Def} L(\bigwedge_{\Delta_1 \in \Delta S'} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \mathbf{next}\, \phi_{\Delta_1}) \vee \bar{\phi}_{\ell,\ell'})) \quad .$$

By definition of set ΔS_ℓ, $L(\Delta S_\ell) =$

$$\bigcap_{\ell' \neq \ell, \ell \to \ell'} L(\Delta S_{\ell,\ell'}) = L(\bigwedge_{\ell' \neq \ell, \ell \to \ell'} \bigwedge_{\Delta_1 \in \Delta S'} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \mathbf{next}\, \phi_{\Delta_1}) \vee \bar{\phi}_{\ell,\ell'}))$$

=[lemma 3.11]

$$L(\bigwedge_{\ell' \neq \ell, \ell \to \ell'} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \mathbf{next} \bigwedge_{\Delta_1 \in \Delta S'} \phi_{\Delta_1}) \vee \bar{\phi}_{\ell,\ell'})) \quad (*)$$

By induction hypothesis,

$$L(\bigwedge_{\Delta_1 \in \Delta S'} \phi_{\Delta_1}) = L(\Delta S_{\ell'})$$

so

$$\bigwedge_{\Delta_1 \in \Delta S'} \phi_{\Delta_1} \sim \phi_{\ell'} \quad ;$$

hence

$$(*) = L(\bigwedge_{\ell' \neq \ell, \ell \to \ell'} \phi_{\ell,\ell} \quad \mathcal{U} \quad ((\phi_{\ell,\ell'} \wedge \mathbf{next}\, \phi_{\ell'}) \vee \bar{\phi}_{\ell,\ell'})) = L(\phi_\ell) \quad .$$

q.e.d.

5.4 Transformation of LSTD specifications

We will next investigate how the semantics of LSTD–specifications behaves under syntactic transformations of (particular diagrams of) the specification.

As preparation, the next lemma contains an important observation.

Lemma 5.4 (Negation–free characterization of LSTD) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–specification ΔS.*

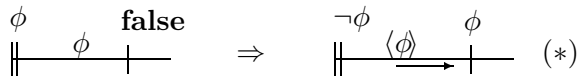
Then the characterizing formula $\phi_{\Delta SPEC}$ of ΔS , constructed according to 5.3 , is in negation–normal form (as defined in definition 3.11).

The proof of lemma 5.4 follows immediately from the constructions used to define formula $\phi_{\Delta SPEC}$. The consequence is that lemma 3.9 can be applied, which states that for a formula in negation normal–form, replacing a sub–formula by another stronger sub–formula, yields a stronger formula.

These facts can be exploited to formulate a number of weakening rules about LSTD–diagrams. In the following, we will treat LSTD–phases, LSTD–bodies and LSTD–diagrams in a way as if they were merely graphical abbreviations of their characterizing formulas.

In particular, we use the notion of *implication* between LSTD objects, as well as the notion of tautology; an example is given next.

Example 5.3 (LSTD–diagram tautology) *We use for LSTD–diagrams the notation of implication introduced for temporal logic formulas (cf. examples 5.1 and 5.2)*



(*) is a tautology because

$$\models \phi_{\alpha\Delta_{\mathbf{always}\ \phi}} \to \phi_{\alpha\Delta_{\mathbf{eventually}\ \phi}} \quad (**)$$

is a temporal logic tautology.

An important point to note about example 5.3 is the fact, that the two LSTD–diagram shown in the implication (*) are *not* structurally similar, at least not in an obvious way. The fact that the diagram on the right side of the implication follows from the diagram on the left side of the implication is based on an analysis of the semantics of these diagrams. On the level of the equivalent reformulation of the implication in terms of the temporal logic characterization of the diagrams (**), the fact that (**) is a tautology follows immediately from the definition of the temporal logic operators **always** and **eventually** .

The rest of this section considers the case, where two LSTD–diagrams are structurally similar. In this case, implication between LSTD–diagrams can often be decided based on "local" conditions, concerning relations between corresponding Boolean formulas occurring in the diagrams.

Implication tautologies between LSTD–diagrams (and LSTD–fragments, e.g. LSTD–phases) will be presented in the form of *rules*, which are displayed in the form

Rule ⟨name⟩ :

$$\begin{array}{ll} \ldots_1 & \text{(premise–1)} \\ \underline{\ldots_2} & \text{(premise–2)} \\ \ldots & \text{(conclusion)} \end{array}$$

i.e. the premise–1 ... premise–k (here shown for $k = 2$) are shown on top of the conclusion, separated from the conclusion by a horiontal line.

Logically related premises can be grouped on a single line, where the individual premises are separated by a , (comma); so the displayed rule can also be denoted in the equivalent form

Rule ⟨name⟩ :

$$\begin{array}{ll} \underline{\ldots_1 \quad , \quad \ldots_2} & \text{(premise)} \\ \ldots & \text{(conclusion)} \end{array}$$

The displayed rule is equivalent to a theorem (or lemma) ⟨name⟩ which says:

T–⟨name⟩ :

Assume that $\ldots_1$ and $\ldots_2$ holds. (premise)

Then... holds. (conclusion)

Rule ⟨name⟩ has to be proven as if it were formulated as theorem T–⟨name⟩.

5.4.1 Transformation of LSTD–phases

The next rule is a first example.

Rule 5.1 (LSTD–phase weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–phase ΔE,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{|}{\underline{\quad|\quad}}}}\langle\phi_3\rangle$$

Consider further a similar LSTD–phase $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi'_2}{\underset{\phi'_1}{\overset{|}{\underline{\quad|\quad}}}}\langle\phi_3\rangle$$

Then the following rule holds:

Rule LSTD–PW–1:

$$\frac{\begin{array}{ll} \phi_1 \Rightarrow \phi'_1 \quad , \quad \phi_2 \Rightarrow \phi'_2 & \textit{(premise–1)} \\ (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) & \\ \sim (\phi_3 \wedge \neg\phi'_1 \wedge \neg\phi'_2) & \textit{(premise–2)} \end{array}}{\Delta E \Rightarrow \Delta E' \qquad \textit{(conclusion)}}$$

Proof of rule 5.1 . We have to prove that under the premises of rule 5.1,

$$\overset{\phi_2}{\underset{\phi_1}{\overset{|}{\underline{\quad|\quad}}}}\langle\phi_3\rangle \;\Rightarrow\; \overset{\phi'_2}{\underset{\phi'_1}{\overset{|}{\underline{\quad|\quad}}}}\langle\phi_3\rangle \qquad (*)$$

is a tautology.

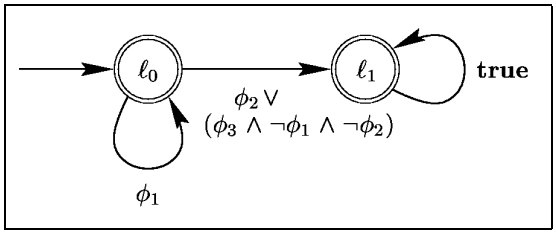
This can be established by showing that on the level of the characterizing temporal logic formulas,

$$\phi_1 \textbf{ unless } (\phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)) \Rightarrow \phi_1' \textbf{ unless } (\phi_2' \vee (\phi_3 \wedge \neg\phi_1' \wedge \neg\phi_2'))$$

is a tautology.

Alternatively, we can investigate the semantics of the associated (partially ordered) Symbolic Automaton.

By definition 5.3, $\mathcal{A}_{\Delta E}$ is defined (graphically depicted) as



Automaton $\mathcal{A}'_{\Delta E}$ has the same structure, where ϕ_1' replaces ϕ_1, and ϕ_2' replaces ϕ_2.

By premise–1 and premise–2 ,

$$\phi_1 \Rightarrow \phi_1'$$
$$\text{and} \quad \phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) \Rightarrow \phi_2' \vee (\phi_3 \wedge \neg\phi_1' \wedge \neg\phi_2')$$

Hence, by lemma 3.1, $L(\mathcal{A}_{\Delta E}) \subseteq L(\mathcal{A}'_{\Delta E})$, so (*) follows.

Rule 5.1 has two variants, which cover most common application situations.

Rule 5.2 (LSTD–phase weakening, variant 1) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–phases ΔE,*

$$\Delta E \equiv_{Def} \frac{\phi_1}{\phi_2}$$

and $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi'_2}{\underset{\phi'_1}{\dashv\!\vdash}}$$

Then the following rule holds:

Rule LSTD–PW–1–v1:

$$\frac{\phi_1 \Rightarrow \phi'_1 \quad , \quad \phi_2 \Rightarrow \phi'_2}{\Delta E \Rightarrow \Delta E'} \quad \begin{array}{l} \textit{(premise–1)} \\ \textit{(conclusion)} \end{array}$$

This rule holds, because it is a special case of rule 5.1 . Note that premise–2 of rule 5.1 follows immediately from the fact, that an omitted exit condition is by definition (by default) equivalent to **false**.

Rule 5.3 (LSTD–phase weakening, variant 2) *Assume an assertion language $A\mathcal{L}$, a set V of variables and LSTD–phases ΔE,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\dashv\!\vdash\!\langle\phi_3\rangle}}$$

and $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi'_2}{\underset{\phi'_1}{\dashv\!\vdash\!\langle\phi_3\rangle}}$$

Then the following rule holds:

Rule LSTD–PW–1–v2:

$$\frac{\begin{array}{c} \phi_1 \Rightarrow \phi'_1 \quad , \quad \phi_2 \Rightarrow \phi'_2 \\ \phi_3 \Rightarrow (\phi_1 \vee \phi_2) \end{array}}{\Delta E \Rightarrow \Delta E'} \quad \begin{array}{l} \textit{(premise–1)} \\ \textit{(premise–2)} \\ \textit{(conclusion)} \end{array}$$

This rule also holds, because it is a special case of rule 5.1 . Note that premise–2 of rule 5.1 follows from the fact, that

$$\frac{\phi_1 \Rightarrow \phi_1' \quad , \quad \phi_2 \Rightarrow \phi_2' \qquad \phi_3 \Rightarrow (\phi_1 \vee \phi_2)}{(\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) \sim \mathbf{false} \quad \text{and} \quad (\phi_3 \wedge \neg\phi_1' \wedge \neg\phi_2') \sim \mathbf{false}}$$

Another variant of rule 5.1 can be built for phases with liveness requirement.

Rule 5.4 (LSTD–phase weakening, additional liveness requirement) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–phase ΔE,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1 \longrightarrow}{\mid \langle\phi_3\rangle}}$$

Consider further a similar LSTD–phase $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi_2'}{\underset{\phi_1' \longrightarrow}{\mid \langle\phi_3\rangle}}$$

Then the following rule holds:

Rule LSTD–PW–1':

$$\frac{\begin{array}{ll} \phi_1 \Rightarrow \phi_1' \quad , \quad \phi_2 \Rightarrow \phi_2' & \textit{(premise–1)} \\ (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) & \\ \sim (\phi_3 \wedge \neg\phi_1' \wedge \neg\phi_2') & \textit{(premise–2)} \end{array}}{\Delta E \Rightarrow \Delta E' \qquad \textit{(conclusion)}}$$

The proof of this rule is literally the same as for rule 5.1 .

It is obvious that omitting the liveness requirement of a phase weakens the phase requirement.

Rule 5.5 (LSTD–phase weakening, omitting liveness requirement) *Assume an assertion language $A\mathcal{L}$, a set V of variables and LSTD–phases ΔE,*

$$\Delta E \equiv_{Def} \xrightarrow{\phi_1} \overset{\phi_2}{|} \langle\phi_3\rangle$$

and $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{|}} \langle\phi_3\rangle$$

Then the following rule holds (rule without premise):

Rule LSTD–PW–2:

$$\frac{}{\Delta E \Rightarrow \Delta E'} \quad \textit{(conclusion)}$$

It remains to consider what is the effect of a transformation (in particual weakening) of the exit condition of a LSTD–phase. This is answered by the next rules.

Rule 5.6 (LSTD–phase exit condition weakening) *Assume an assertion language* $A\mathcal{L}$*, a set* V *of variables and LSTD–phases* ΔE*,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{|}} \langle\phi_3\rangle$$

and $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{|}} \langle\phi'_3\rangle$$

Then the following rule holds:

Rule LSTD–PW–EC:

$$\frac{\phi_3 \Rightarrow \phi'_3 \quad \textit{(premise)}}{\Delta E \Rightarrow \Delta E' \quad \textit{(conclusion)}}$$

Rule 5.7 (LSTD–phase exit condition weakening, additional liveness requirement) *Assume an assertion language $A\mathcal{L}$, a set V of variables and LSTD–phases ΔE,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\longrightarrow}} \langle \phi_3 \rangle$$

and $\Delta E'$,

$$\Delta E' \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\longrightarrow}} \langle \phi_3' \rangle$$

Then the following rule holds:

Rule LSTD–PW–EC':

$$\frac{\phi_3 \Rightarrow \phi_3' \qquad \textit{(premise)}}{\Delta E \Rightarrow \Delta E' \qquad \textit{(conclusion)}}$$

The proof of rule 5.6 and rule 5.7 is the same as for rule 5.1.

The automaton $\mathcal{A}_{\Delta E'}$ has the same structure as automaton $\mathcal{A}_{\Delta E}$, where ϕ_3' replaces ϕ_3.

The premise

$$\phi_3 \Rightarrow \phi_3'$$
$$\text{implies} \quad \phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) \Rightarrow \phi_2 \vee (\phi_3' \wedge \neg\phi_1 \wedge \neg\phi_2)$$

Hence, by lemma 3.1, $L(\mathcal{A}_{\Delta E}) \subseteq L(\mathcal{A}_{\Delta E'})$, so the conclusion of the rule follows in both cases from the premise.

Another type of rule considers the case, where two phases are "active" at the same time. This is important, because a LSTD–specification contains in general more than one diagram.

We will use the notation

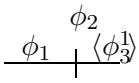
$$\frac{\ldots}{\{\Delta E^1, \Delta E^2\} \Rightarrow \Delta E} \qquad \textit{(conclusion)}$$

to denote that fact that the conjunction of ΔE_1 and ΔE_2 implies the LSTD–phase ΔE.

On the visual level, the simultaneous effect of two LSTD–objects (phases or diagrams) working in cooperation is called *superposition*. Superposition assumes, that activation of the phases occurs at the same time, which must be ensured by the context of the phases.

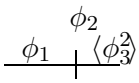
Rule 5.8 (LSTD–phase superposition) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–phase ΔE^1,*

$$\Delta E^1 \equiv_{Def}$$



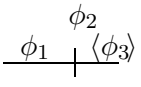
Consider further similar LSTD–phases ΔE^2,

$$\Delta E^2 \equiv_{Def}$$



and

$$\Delta E \equiv_{Def}$$



Then the following rule holds:

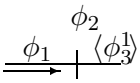
Rule LSTD–SUP:

$$\frac{\phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3 \qquad \textit{(premise)}}{\{\Delta E^1, \Delta E^2\} \Rightarrow \Delta E \qquad \textit{(conclusion)}}$$

Under the premise of this rule, ΔE is called a superposition of the phase.

Rule 5.9 (LSTD–phase superposition, with additional liveness requirement) *Assume an assertion language $A\mathcal{L}$, a set V of variables and an LSTD–phase ΔE^1,*

$$\Delta E^1 \equiv_{Def}$$



Consider further similar LSTD–phases ΔE^2,

$$\Delta E^2 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{\langle\phi_3^2\rangle}{\dashv\!\!\vdash}}}$$

and

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{\langle\phi_3\rangle}{\dashv\!\!\vdash}}}$$

Then the following rule holds:

Rule LSTD–SUP':

$$\frac{\phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3 \qquad \textit{(premise)}}{\{\Delta E^1, \Delta E^2\} \Rightarrow \Delta E \qquad \textit{(conclusion)}}$$

For the proof of rule 5.8 and of rule 5.9 we have to show that

$$L(\mathcal{A}_{\Delta E^1}) \cap L(\mathcal{A}_{\Delta E^2}) \subseteq L(\mathcal{A}_{\Delta E}) \qquad (*)$$

i.e. for all computations $\sigma \in Comp(V)$: (LHS) $\Rightarrow$ (RHS), where

$$\sigma \in L(\mathcal{A}_{\Delta E^1}) \wedge \quad \sigma \in L(\mathcal{A}_{\Delta E^2}) \qquad \text{(LHS)}$$
$$\sigma \in L(\mathcal{A}_{\Delta E}) \qquad \text{(RHS)}$$

We have to consider the following cases:

- case 1: $\sigma \models \mathbf{always}\,\phi_1$. In case of rule 5.8, σ is accepted by $\mathcal{A}_{\Delta E}$, and (RHS) holds. In case of rule 5.9, we must consider further sub–cases.

- case 1.1: $\exists k{\geq}0$. $\sigma(k) \models \phi_2$. Then σ is accepted by $\mathcal{A}_{\Delta E}$, and (RHS) holds.

- case 1.2: $\sigma \models \mathbf{always}\,\neg\phi_2$. Since $\sigma \models \mathbf{always}\,\phi_1$, this implies that

$$\sigma \models \mathbf{always}\,\neg(\phi_2 \vee (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2))$$

Hence, σ is not accepted by $\mathcal{A}_{\Delta E}$. Furthermore, σ is not accepted by $\mathcal{A}_{\Delta E^1}$ for the same reason (ϕ_3^1 replaces ϕ_3 in the argument). Hence, (LHS) does not hold, and the implication (LHS) $\Rightarrow$ (RHS) is true.

- case 2: $\neg(\sigma \models \textbf{always}\, \phi_1)$, i.e.

$$\exists k{\geq}0\ .\ (\forall i, 0{\leq}i < k : \sigma(i) \models \phi_1) \wedge \sigma(k) \models \neg\phi_1$$

- case 2.1: $\exists i{\leq}k\ .\ \sigma(i) \models \phi_2$. Then σ is accepted by $\mathcal{A}_{\Delta E}$, and (RHS) holds.
- case 2.2: $\forall i{\leq}k : \sigma(i) \models \neg\phi_2$. In this case we have to consider further sub–cases:
- case 2.2.1: $\sigma(k) \models (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)$. Then σ is accepted by $\mathcal{A}_{\Delta E}$, and (RHS) holds.
- case 2.2.2: $\sigma(k) \models \neg(\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)$. Then σ is not accepted by $\mathcal{A}_{\Delta E}$. By premise, $\neg\phi_3$ implies that either $\neg\phi_3^1$ or $\neg\phi_3^2$ holds in step k. Assume w.l.o.g that $\neg\phi_3^1$ holds in step k, then σ is not accepted by $\mathcal{A}_{\Delta E^1}$. Hence, (LHS) does not hold, and the implication (LHS) $\Rightarrow$ (RHS) is true.

5.4.2 Transformation of LSTD–bodies

An LSTD–body Δ consists of a non–empty sequence of phases,

$$\Delta \equiv_{Def} \Delta E_1, \ldots, \Delta E_k$$

We will next investigate the effect of the transformation of a phase within an LSTD–body. This will be done in two steps:

1. Transformation on the head of an LSTD–body is considered;
2. Transformation of an "inner" part (phase) of an LSTD–body is considered.

Transformation of LSTD–body head phase. The main result of this section is that the monotonicity rules derived for single LSTD–phases can be extended to the level of LSTD–bodies.

First, for each rule presented in the previous section, there is a corresponding rule on the level of LSTD–bodies, where the *head* of the body (i.e. the

"leftmost" phase) is weakened. Moreover, the argument of the corresponding proof of the rule remains essentially the same.

Rule 5.10 (LSTD–head–phase weakening) *Assume the assumptions of rule 5.1 and an LSTD–phase* Δ_1 *over* V *. In particular,*

$$\Delta E \equiv_{Def} \begin{array}{c} \phi_2 \\ \hline \phi_1 \mid \langle\phi_3\rangle \end{array}$$

and

$$\Delta E' \equiv_{Def} \begin{array}{c} \phi_2' \\ \hline \phi_1' \mid \langle\phi_3\rangle \end{array}$$

Then the following rule holds:

Rule LSTD–HPW–1:

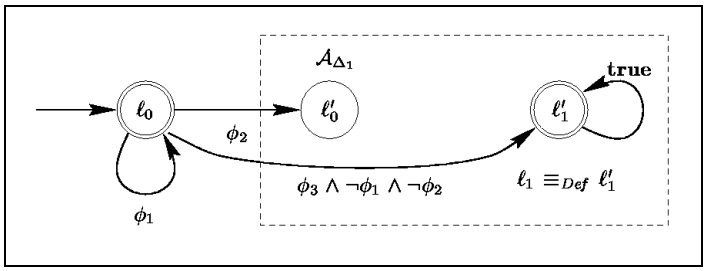
$$\frac{\begin{array}{ll} \phi_1 \Rightarrow \phi_1' \quad , \quad \phi_2 \Rightarrow \phi_2' & \textit{(premise–1)} \\ (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) & \\ \sim (\phi_3 \wedge \neg\phi_1' \wedge \neg\phi_2') & \textit{(premise–2)} \end{array}}{\Delta E\ \Delta_1 \Rightarrow \Delta E'\ \Delta_1 \qquad \textit{(conclusion)}}$$

Proof of rule 5.10 . Assume the premises of rule 5.10.

We investigate the semantics of the associated (partially ordered) Symbolic Automata of the LSTD–bodies

$$\begin{array}{lll} \Delta^1 & \equiv_{Def} & \Delta E\ \Delta_1 \\ \Delta^2 & \equiv_{Def} & \Delta E'\ \Delta_1 \end{array}$$

By definition 5.3, $\mathcal{A}_{\Delta^1}$ is defined (graphically depicted) as



Automaton $\mathcal{A}_{\Delta^2}$ has the same structure, where ϕ'_1 replaces ϕ_1, and ϕ'_2 replaces ϕ_2.

By premise–1 and premise–2 , lemma 3.1 applies. Hence $L(\mathcal{A}_{\Delta^1}) \subseteq L(\mathcal{A}_{\Delta^2})$, and the conclusion of the rule follows.

We omit the reformulation of the rules 5.2 and 5.3. The next two rules consider LSTD–head–phase weakening with additional liveness requirements, and the omission of the liveness requirement from a head phase.

Rule 5.11 (LSTD–head–phase weakening, additional liveness requirement) *Assume the assumptions of 5.4 and an LSTD–phase Δ_1 over V . In particular,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\Longrightarrow\!\!\mid}}\langle\phi_3\rangle$$

and

$$\Delta E' \equiv_{Def} \overset{\phi'_2}{\underset{\phi'_1}{\Longrightarrow\!\!\mid}}\langle\phi_3\rangle$$

Then the following rule holds:

Rule LSTD–HPW–1':

$$\frac{\begin{array}{cc}\phi_1 \Rightarrow \phi'_1 \quad , \quad \phi_2 \Rightarrow \phi'_2 & \textit{(premise–1)}\\ (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) & \\ \sim (\phi_3 \wedge \neg\phi'_1 \wedge \neg\phi'_2) & \textit{(premise–2)}\end{array}}{\Delta E\ \Delta_1 \Rightarrow \Delta E'\ \Delta_1 \qquad \textit{(conclusion)}}$$

The proof of this rule is the same as for rule 5.10 and exploits again the monotonicity argument of lemma 3.1 .

For the same reason, it is obvious that omitting the liveness requirement of a phase weakens the phase requirement.

Rule 5.12 (LSTD–head–phase weakening, omitting liveness requirement) *Assume the premises of rule 5.5 , in particular*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\longrightarrow}} \langle \phi_3 \rangle$$

and

$$\Delta E' \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\text{---}}} \langle \phi_3 \rangle$$

and an LSTD–phase Δ_1 over V.

Then the following rule holds (rule without premise):

Rule LSTD–HPW–2:

$$\frac{}{\Delta E\ \Delta_1 \Rightarrow \Delta E'\ \Delta_1} \qquad \textit{(conclusion)}$$

Next, the rules about LSTD–phase exit condition weakening are extended.

Rule 5.13 (LSTD–head–phase exit condition weakening) *Assume the premises of rule 5.6 , in particular*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\text{---}}} \langle \phi_3 \rangle$$

and

$$\Delta E' \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\text{---}}} \langle \phi_3' \rangle$$

and an LSTD–phase Δ_1 over V .

Then the following rule holds:

Rule LSTD–HPW–EC:

$$\frac{\phi_3 \Rightarrow \phi_3' \qquad \textit{(premise)}}{\Delta E\ \Delta_1 \Rightarrow \Delta E'\ \Delta_1 \quad \textit{(conclusion)}}$$

Rule 5.14 (LSTD–head–phase exit condition weakening, additional liveness requirement) *Assume the premises of rule 5.7 , in particular*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\Longrightarrow\!\!|}}\ \langle\phi_3\rangle$$

and

$$\Delta E' \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\Longrightarrow\!\!|}}\ \langle\phi_3'\rangle$$

and an LSTD–phase Δ_1 over V .

Then the following rule holds:

Rule LSTD–HPW–EC':

$$\frac{\phi_3 \Rightarrow \phi_3' \qquad \textit{(premise)}}{\Delta E\ \Delta_1 \Rightarrow \Delta E'\ \Delta_1 \quad \textit{(conclusion)}}$$

Next, the rules for LSTD–phase superposition are extended.

Rule 5.15 (LSTD–head–phase superposition) *Assume the premises of rule 5.8, in particular*

$$\Delta E^1 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{-\!\!|}}\ \langle\phi_3^1\rangle$$

and similar LSTD–phases ΔE^2,

$$\Delta E^2 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{-\!\!|}}\ \langle\phi_3^2\rangle$$

and ΔE*,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{|}{\underset{|}{\langle \phi_3 \rangle}}}}$$

and an LSTD–phase Δ_1 *over* V*. Then the following rule holds :*

Rule LSTD–HP–SUP:

$$\frac{\phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3 \qquad \textit{(premise)}}{\{\Delta E^1\ \Delta_1, \Delta E^2\ \Delta_1\} \Rightarrow \Delta E\ \Delta_1 \qquad \textit{(conclusion)}}$$

Rule 5.16 (LSTD–head–phase superposition, with additional liveness–requirement) *Assume the premises of rule 5.9, in particular*

$$\Delta E^1 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{|}{\underset{|}{\langle \phi_3^1 \rangle}}}}$$

and similar LSTD–phases ΔE^2*,*

$$\Delta E^2 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{|}{\underset{|}{\langle \phi_3^2 \rangle}}}}$$

and ΔE*,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\overset{|}{\underset{|}{\langle \phi_3 \rangle}}}}$$

and an LSTD–phase Δ_1 *over* V*. Then the following rule holds :*

Rule LSTD–HP–SUP':

$$\frac{\phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3 \qquad \textit{(premise)}}{\{\Delta E^1\ \Delta_1, \Delta E^2\ \Delta_1\} \Rightarrow \Delta E\ \Delta_1 \qquad \textit{(conclusion)}}$$

The proof of these rules is similar to the proofs of rules 5.8 and 5.9, with small adaptations.

Let

$$\begin{array}{lll} \Delta^1 & \equiv_{Def} & \Delta E^1\ \Delta_1 \\ \Delta^2 & \equiv_{Def} & \Delta E^2\ \Delta_1 \\ \Delta & \equiv_{Def} & \Delta E\ \Delta_1 \end{array}$$

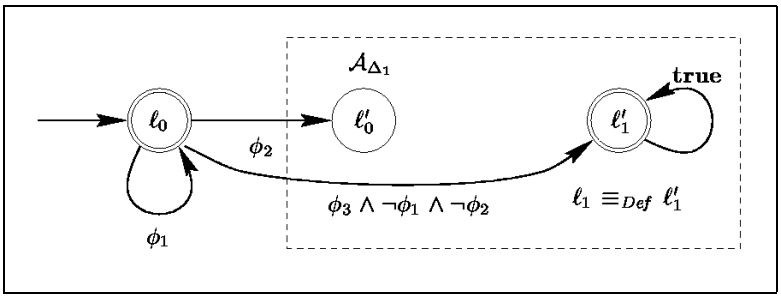
We have to show that

$$L(\mathcal{A}_{\Delta^1}) \cap L(\mathcal{A}_{\Delta^2}) \subseteq L(\mathcal{A}_{\Delta}) \qquad (*)$$

i.e. for all computations $\sigma \in Comp(V)$: (LHS) $\Rightarrow$ (RHS), where

$$\begin{array}{lll} \sigma \in L(\mathcal{A}_{\Delta^1}) \wedge & \sigma \in L(\mathcal{A}_{\Delta^2}) & \text{(LHS)} \\ \sigma \in L(\mathcal{A}_{\Delta}) & & \text{(RHS)} \end{array}$$

Recall that the SA $\mathcal{A}_\Delta$ is (graphically depicted in case of rule 5.15) defined as



We have to consider the following cases:

- case 1: $\sigma \models$ **always** ϕ_1 . In case of rule 5.15, σ is accepted by $\mathcal{A}_\Delta$, and (RHS) holds. In case of rule 5.16, we must consider further sub–cases.
- case 1.1: $\exists k{\geq}0\ .\ \sigma(k) \models \phi_2 \wedge \sigma^{(k+1)} \in L(\mathcal{A}_{\Delta_1})$. In this case, σ is accepted by $\mathcal{A}_\Delta$, i.e.(RHS) holds.
- case 1.2: $\forall k{\geq}0\ :\ \sigma(k) \models \neg\phi_2 \vee \sigma^{(k+1)} \notin L(\mathcal{A}_\Delta)$. Consider LHS, automaton $\mathcal{A}_{\Delta^1}$: Since $\sigma \models$ **always** ϕ_1, the only possibility that σ could be accepted by $\mathcal{A}_\Delta$ is by taking a transition from ℓ_0 to ℓ'_0 in some step k, with subsequent acceptance by $\mathcal{A}_{\Delta_1}$. Since this possibility is excluded in this case, (LHS) does not hold.

- case 2: $\neg(\sigma \models \textbf{always}\, \phi_1)$, i.e.

$$\exists k_0 \geq 0 \,.\, (\forall i, 0 \leq i < k_0 : \sigma(i) \models \phi_1) \wedge \sigma(k_0) \models \neg\phi_1$$

  (k_0 is the first step where $\sigma(k_0) \models \neg\phi_1$.)

- case 2.1: $\exists i \leq k_0 \,.\, \sigma(i) \models \phi_2 \wedge \sigma^{(i+1)} \in L(\mathcal{A}_{\Delta_1})$. In this case, σ is accepted by $\mathcal{A}_\Delta$, i.e.(RHS) holds.

- case 2.2: $\forall i \leq k_0 \,.\, \sigma(i) \models \neg\phi_2 \vee \sigma^{(i+1)} \notin L(\mathcal{A}_{\Delta_1})$.

- case 2.2.1: $\sigma(k_0) \models (\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)$. Then σ is accepted by $\mathcal{A}_\Delta$, and (RHS) holds.

- case 2.2.2: $\sigma(k_0) \models \neg(\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2)$. Since $\sigma(k_0) \models \neg\phi_2 \vee \sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1})$, σ is not accepted by $\mathcal{A}_\Delta$. By premise, $\neg\phi_3$ implies that either $\neg\phi_3^1$ or $\neg\phi_3^2$ holds in step k_0. Assume w.l.o.g that $\neg\phi_3^1$ holds in step k_0. Since $\sigma(k_0) \models \neg\phi_1$ and $\sigma(k_0) \models \neg(\phi_3^1 \wedge \neg\phi_1 \wedge \neg\phi_2)$, σ is not accepted by $\mathcal{A}_{\Delta^1}$. Hence, (LHS) does not hold, and the implication (LHS) $\Rightarrow$ (RHS) is true.

An important extension of the rules 5.15 and 5.16 is possible in the case that the phase ΔE is *deterministic*, i.e. $\phi_1 \Rightarrow \neg\phi_2$.

Rule 5.17 (LSTD–deterministic–head–phase superposition) *Assume the premises of rule 5.8, in particular*

$$\Delta E^1 \equiv_{Def} \begin{array}{c|c} \phi_1 & \phi_2 \\ \hline & \langle \phi_3^1 \rangle \end{array}$$

and similar LSTD–phases ΔE^2*,*

$$\Delta E^2 \equiv_{Def} \begin{array}{c|c} \phi_1 & \phi_2 \\ \hline & \langle \phi_3^2 \rangle \end{array}$$

and ΔE*,*

$$\Delta E \equiv_{Def} \begin{array}{c|c} \phi_1 & \phi_2 \\ \hline & \langle \phi_3 \rangle \end{array}$$

and LSTD–phases Δ_1^1, Δ_1^2 *and* Δ_1 *over* V*. Then the following rule holds :*

Rule LSTD–DHP–SUP:

$$\frac{\begin{array}{cc} \phi_1 \Rightarrow \neg\phi_2 & \textit{(premise–1)} \\ \{\Delta_1^1, \Delta_1^2\} \Rightarrow \Delta_1 & \textit{(premise–2)} \\ \phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3 & \textit{(premise–3)} \end{array}}{\{\Delta E^1\ \Delta_1^1, \Delta E^2\ \Delta_1^2\} \Rightarrow \Delta E\ \Delta_1 \quad \textit{(conclusion)}}$$

Rule 5.18 (LSTD–deterministic–head–phase superposition, with add. liveness req.) *Assume the premises of rule 5.9, in particular*

$$\Delta E^1 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\Longrightarrow\!\!\mid}}\ \langle\phi_3^1\rangle$$

and similar LSTD–phases ΔE^2*,*

$$\Delta E^2 \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\longrightarrow\!\!\mid}}\ \langle\phi_3^2\rangle$$

and ΔE*,*

$$\Delta E \equiv_{Def} \overset{\phi_2}{\underset{\phi_1}{\Longrightarrow\!\!\mid}}\ \langle\phi_3\rangle$$

and LSTD–phases Δ_1^1, Δ_1^2 *and* Δ_1 *over* V*. Then the following rule holds :*

Rule LSTD–DHP–SUP':

$$\frac{\begin{array}{cc} \phi_1 \Rightarrow \neg\phi_2 & \textit{(premise–1)} \\ \{\Delta_1^1, \Delta_1^2\} \Rightarrow \Delta_1 & \textit{(premise–2)} \\ \phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3 & \textit{(premise–3)} \end{array}}{\{\Delta E^1\ \Delta_1^1, \Delta E^2\ \Delta_1^2\} \Rightarrow \Delta E\ \Delta_1 \quad \textit{(conclusion)}}$$

The proof of these rules is similar to the proof of rules 5.15 and 5.16; in particular, similar abbreviations are used:

$$\begin{array}{lll} \Delta^1 & \equiv_{Def} & \Delta E^1\ \Delta_1^1 \\ \Delta^2 & \equiv_{Def} & \Delta E^2\ \Delta_1^2 \\ \Delta & \equiv_{Def} & \Delta E\ \Delta_1 \end{array}$$

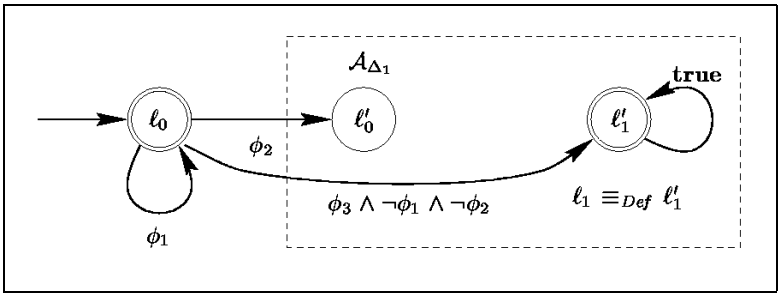
Again, we have to show that

$$L(\mathcal{A}_{\Delta^1}) \cap L(\mathcal{A}_{\Delta^2}) \subseteq L(\mathcal{A}_{\Delta}) \qquad (*)$$

i.e. for all computations $\sigma \in Comp(V)$: (LHS) $\Rightarrow$ (RHS), where

$$\sigma \in L(\mathcal{A}_{\Delta^1}) \wedge \sigma \in L(\mathcal{A}_{\Delta^2}) \qquad \text{(LHS)}$$
$$\sigma \in L(\mathcal{A}_{\Delta}) \qquad \text{(RHS)}$$

The SA $\mathcal{A}_\Delta$ is (graphically depicted in case of rule 5.17) defined as



We have to consider the following cases:

- case 1: $\sigma \models$ **always** ϕ_1 . In case of rule 5.17, σ is accepted by $\mathcal{A}_\Delta$, and (RHS) holds. In case of rule 5.18, σ is not accepted by $\mathcal{A}_{\Delta_1}$, so (LHS) does not hold.

- case 2: $\neg(\sigma \models$ **always** $\phi_1)$, i.e.

$$\exists k_0 \geq 0\ .\ (\forall i, 0 \leq i < k_0 : \sigma(i) \models \phi_1) \wedge \sigma(k_0) \models \neg\phi_1$$

(k_0 is the first step where $\sigma(k_0) \models \neg\phi_1$.)

If $\sigma \in L(\mathcal{A}_\Delta)$, then RHS holds. Hence, assume that $\sigma \notin L(\mathcal{A}_\Delta)$. Then

$$\sigma(k_0) \models \neg(\phi_3 \wedge \neg\phi_1 \wedge \neg\phi_2) \qquad (1), \quad \text{and}$$
$$\sigma(k_0) \models \neg\phi_2 \vee \sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1}) \qquad (2)$$

By case 2, $\sigma(k_0) \models \phi_1$, and by premise, $\sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1})$ implies $\sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1^1})$ or $\sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1^2})$. Thus, the following facts hold:

$$\sigma(k_0) \models \neg\phi_3 \vee \phi_2 \qquad (1'), \quad \text{and}$$
$$\sigma(k_0) \models \neg\phi_2 \vee \sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1^1}) \vee \sigma^{(k_0+1)} \notin L(\mathcal{A}_{\Delta_1^2}) \qquad (2')$$

If $\sigma(k_0) \models \neg\phi_2$, then by (1') $\sigma(k_0) \models \neg\phi_3$. By premise $\phi_3^1 \wedge \phi_3^2 \Rightarrow \phi_3$, which means that either $\sigma(k_0) \models \neg\phi_3^1$ or $\sigma(k_0) \models \neg\phi_3^2$, so either $\sigma \notin L(\mathcal{A}_{\Delta^1})$ or $\sigma \notin L(\mathcal{A}_{\Delta^2})$.

If $\sigma(k_0) \models \phi_2$, then the same conclusion follows from (2'). q.e.d.

Transformation of inner phases of an LSTD–body. The rules considered so far can be further extended to cover the case, where an "inner" phase of an LSTD–body is weakened.

In general, an LSTD–body Δ consists of a non–empty sequence of phases,

$$\Delta \equiv_{Def} \Delta E_1 \ldots \Delta E_j \ldots \Delta E_k \qquad (*)$$

for some $k \geq 1$. We would like to extend the rules about LSTD–head–phase weakening to the case, where $\Delta E_j \Rightarrow \Delta E'_j$.

This can be done by the following argument: Assume that in (*), $2 \leq j \leq k$. Then Δ can be presented in the form

$$\Delta \equiv_{Def} \Delta E_1 \ldots \Delta E_{j-1} \Delta_1$$

where

$$\Delta_1 \equiv_{Def} \Delta E_j, \ldots, \Delta E_k \qquad (**)$$

If a rule about LSTD–head–phase–weakening applies due to the similarity of ΔE_j and $\Delta E'_j$, then we can possibly derive

$$\Delta'_1 \equiv_{Def} \Delta E'_j \ldots \Delta E_k$$

from Δ_1, defined in (**). The next two rules show that this can be used in turn to derive

$$\Delta' \equiv_{Def} \Delta E_1 \ldots \Delta E'_j \ldots \Delta E_k$$

from Δ.

Rule 5.19 (LSTD–tail weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , an LSTD–phase ΔE and LSTD–bodies Δ_1 and Δ'_1 over V.*

Then the following rule holds:

Rule LSTD–TW:

$$\frac{\Delta_1 \Rightarrow \Delta'_1 \qquad \textit{(premise)}}{\Delta E\ \Delta_1 \Rightarrow \Delta E\ \Delta'_1 \qquad \textit{(conclusion)}}$$

Proof of rule 5.19 . Let

$$\Delta^1 \equiv_{Def} \Delta E\ \Delta_1$$

and

$$\Delta^2 \equiv_{Def} \Delta E\ \Delta'_1 \quad .$$

By definition 5.6 of ϕ_{Δ^1},

$$\phi_{\Delta^1} \equiv_{Def} \Phi[\phi_{\Delta_1}/u]$$

Where the formula–scheme Φ (cf. definition 3.10) is defined by

$$\Phi \quad \equiv_{Def} \quad \phi_1\, \mathcal{U}\, (\phi_2 \wedge u \vee (\phi_2 \wedge \neg\phi_1 \wedge \neg\phi_2))$$

Similary,

$$\phi_{\Delta^2} \equiv_{Def} \Phi[\phi_{\Delta'_1}/u]$$

By premise,

$$\phi_{\Delta_1} \Rightarrow \phi_{\Delta'_1}$$

Hence by lemma 3.9,

$$\begin{aligned} \phi_{\Delta^1} \equiv \quad & \Phi[\phi_{\Delta_1}/u] \\ \Rightarrow \quad & \Phi[\phi_{\Delta'_1}/u] \quad \equiv \phi_{\Delta^2} \end{aligned}$$

the conclusion of rule 5.19 follows.

Rule 5.20 (LSTD–tail–part weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , LSTD–phases $\Delta E_1 \ldots \Delta E_k$ and LSTD–bodies Δ_1 and Δ'_1 over V.*

Let

$$\Delta^1 \equiv_{Def} \Delta E_1 \ldots \Delta E_k \ \Delta_1$$

and

$$\Delta^2 \equiv_{Def} \Delta E_1 \ldots \Delta E_k \ \Delta'_1 \quad .$$

Then the following rule holds:

Rule LSTD–TPW:

$$\frac{\Delta_1 \Rightarrow \Delta'_1 \qquad \textit{(premise)}}{\Delta^1 \Rightarrow \Delta^2 \qquad \textit{(conclusion)}}$$

Proof of rule 5.20 . The proof is by induction on k.

- case $k = 1$: In this case, the conclusion follows from rule 5.19.
- case $k \rightarrow k+1$: Assume that the rule has been proven up to a fixed value of k (Induction premise). By the inductive definition of LSTD–bodies (cf. def. 5.1), we can (uniquely) present

$$\Delta^1 \equiv_{Def} \Delta E_1 \ldots \Delta E_k \Delta E_{k+1} \ \Delta_1$$

as

$$\Delta^1 \equiv_{Def} \Delta E_1 \ldots \Delta E_k \ \Delta_{11}$$

where

$$\Delta_{11} \equiv_{Def} \Delta E_{k+1} \ \Delta_1 \quad ;$$

similarly, we can present

$$\Delta^2 \equiv_{Def} \Delta E_1 \ldots \Delta E_k \Delta E_{k+1} \ \Delta'_1$$

as

$$\Delta^2 \equiv_{Def} \Delta E_1 \ldots \Delta E_k \ \Delta'_{11}$$

where

$$\Delta'_{11} \equiv_{Def} \Delta E_{k+1} \ \Delta'_1 \quad .$$

Then we can derive the conclusion of rule 5.20 as follows:

$$\dfrac{\dfrac{\Delta_1 \Rightarrow \Delta'_1}{\underbrace{\Delta E_{k+1} \ \Delta_1}_{\Delta_{11}} \Rightarrow \underbrace{\Delta E_{k+1} \ \Delta'_1}_{\Delta'_{11}}}}{\Delta^1 \Rightarrow \Delta^2}$$

premise of rule 5.19

Induction premise of this rule

(conclusion)

5.4.3 Transformation of LSTD–diagrams

The last block of LSTD–rules covers the remaining LSTD constructions, which are LSTD–diagrams and LSTD–specifications.

Since these constructions (like all the other constructions of LSTD objects) are "negation–free", the weakening rules can be canonically extended to the level of diagrams and specifications.

Rule 5.21 (LSTD–initial–diagram condition weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas $\phi_1, \phi_2, \phi'_1, \phi'_2$ and an LSTD–body Δ_1 over V.*

Let

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\Vert}\underline{\;\langle\phi_2\rangle\;}\,\Delta_1$$

and

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1'}{\Vert}\underline{\;\langle\phi_2'\rangle\;}\,\Delta_1$$

Then the following rule holds:

Rule LSTD–IDW–1:

$$\frac{\phi_1 \Rightarrow \phi_1' \;,\; \phi_2 \Rightarrow \phi_2'}{\alpha\Delta^1 \Rightarrow \alpha\Delta^2} \qquad \begin{matrix}\textit{(premise)}\\ \textit{(conclusion)}\end{matrix}$$

Proof of rule 5.21 . By lemma 5.2 (characterizing formula $\phi_{\alpha\Delta}$ of LSTD–diagram $\alpha\Delta$),

$$\phi_{\alpha\Delta^1} = \phi_2 \vee (\phi_1 \wedge \textbf{next}\, \phi_{\Delta_1})$$

and

$$\phi_{\alpha\Delta^2} = \phi_2' \vee (\phi_1' \wedge \textbf{next}\, \phi_{\Delta_1})$$

Then

$$\phi_{\alpha\Delta^1} \Rightarrow \phi_{\alpha\Delta^2}$$

follows immediately from the definition of the notion of a temporal logic tautology, so the conclusion of rule 5.21 follows.

Rule 5.22 (LSTD–initial–diagram body weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas ϕ_1, ϕ_2, and LSTD–bodies Δ_1 and Δ_1' over V.*

Let

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\Vert}\!\xrightarrow{\langle\phi_2\rangle} \Delta_1$$

and

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1}{\Vert}\!\xrightarrow{\langle\phi_2\rangle} \Delta_1'$$

Then the following rule holds:

Rule LSTD–IDW–2:

$$\frac{\Delta_1 \Rightarrow \Delta_1' \qquad \textit{(premise)}}{\alpha\Delta^1 \Rightarrow \alpha\Delta^2 \qquad \textit{(conclusion)}}$$

Proof of rule 5.22 . By lemma 5.2 (characterizing formula $\phi_{\alpha\Delta}$ of LSTD–diagram $\alpha\Delta$),

$$\phi_{\alpha\Delta} \equiv_{Def} \Phi[\phi_{\Delta_1}/u]$$

where the formula–scheme Φ (cf. definition 3.10) is defined by

$$\Phi \quad \equiv_{Def} \quad \phi_2 \vee (\phi_1 \wedge \textbf{next}\, u\ )$$

Similary,

$$\phi_{\alpha\Delta^2} \equiv_{Def} \Phi[\phi_{\Delta_1'}/u]$$

By premise,

$$\phi_{\Delta_1} \Rightarrow \phi_{\Delta_1'}$$

Hence by lemma 3.9,

$$\begin{aligned} \phi_{\alpha\Delta^1} \equiv \quad & \Phi[\phi_{\Delta_1}/u] \\ \Rightarrow \quad & \Phi[\phi_{\Delta_1'}/u] \quad \equiv \phi_{\alpha\Delta^2} \end{aligned}$$

the conclusion of rule 5.22 follows.

Rule 5.23 (LSTD–invariant–diagram activation condition weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas ϕ_1, ϕ_1' and an LSTD–body Δ_1 over V.*

Let

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\longmapsto} \Delta_1$$

and

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1'}{\longmapsto} \Delta_1$$

Then the following rule holds:

Rule LSTD–ADW–1:

$$\frac{\phi_1' \Rightarrow \phi_1 \quad \textit{(premise)}}{\alpha\Delta^1 \Rightarrow \alpha\Delta^2 \quad \textit{(conclusion)}}$$

Proof of rule 5.23 . By lemma 5.2 (characterizing formula $\phi_{\alpha\Delta}$ of LSTD–diagram $\alpha\Delta$),

$$\phi_{\alpha\Delta^1} = \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,\phi_{\Delta_1})$$

and

$$\phi_{\alpha\Delta^2} = \textbf{always}\,(\phi_1' \rightarrow \textbf{next}\,\phi_{\Delta_1})$$

We have to show that

$$\phi_{\alpha\Delta^1} \Rightarrow \phi_{\alpha\Delta^2}$$

follows from the definition of the notion of a temporal logic tautology; in particular,

$$\forall \sigma \in Comp(V): \qquad (*)$$
$$\sigma \models \alpha\Delta_1 \rightarrow \sigma \models \alpha\Delta_2$$

Assume that (*) does not hold, i.e. there exists some computation σ_0 such that $\sigma \models \alpha\Delta_1$and:

$$\exists k{\geq}0 \,.\, \sigma_0(k) \models \phi_2' \wedge \sigma_0^{k+1} \models \phi_{\Delta_1}$$

Then, from the premise of rule 5.23 (with the same value of k) it follows that

$$\exists k{\geq}0 \,.\, \sigma_0(k) \models \phi_2 \wedge \sigma_0^{k+1} \models \phi_{\Delta_1}$$

So $\sigma \models \alpha\Delta_1$ does not hold under the assumption that (*) does not hold; hence the conclusion of rule 5.23 follows.

Rule 5.24 (LSTD–invariant–diagram body weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formula ϕ_1, and LSTD–bodies Δ_1, Δ_1' over V.*

Let

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!\!-} \Delta_1$$

and

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!\!-} \Delta_1'$$

Then the following rule holds:

Rule LSTD–ADW–2:

$$\frac{\Delta_1 \Rightarrow \Delta_1' \qquad \textit{(premise)}}{\alpha\Delta^1 \Rightarrow \alpha\Delta^2 \qquad \textit{(conclusion)}}$$

Proof of rule 5.24 . By lemma 5.2 (characterizing formula $\phi_{\alpha\Delta}$ of LSTD–diagram $\alpha\Delta$),

$$\phi_{\alpha\Delta} \equiv_{Def} \Phi[\phi_{\Delta_1}/u]$$

where the formula–scheme Φ (cf. definition 3.10) is defined by

$$\Phi \quad \equiv_{Def} \quad \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,u\,)$$

The remaining part of the proof of this rule is omitted, since it uses the same substitution argument as the proof of the similar rule 5.22.

The next rules extend the rules about superposition to the level of LSTD–diagrams.

Rule 5.25 (LSTD–initial–diagram superposition) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas $\phi_1, \phi_2, \phi_2^1, \phi_2^2$, and an LSTD–body Δ_1 over V.*

Let

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\underset{\langle\phi_2^1\rangle}{\Vdash\!\!\!-\!\!\!-}}\,\Delta_1 \quad ,$$

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1}{\underset{\langle\phi_2^2\rangle}{\Vdash\!\!\!-\!\!\!-}}\,\Delta_1$$

and

$$\alpha\Delta \equiv_{Def} \overset{\phi_1}{\underset{\langle\phi_2\rangle}{\Vdash\!\!\!-\!\!\!-}}\,\Delta_1 \quad .$$

Then the following rule holds:

Rule LSTD–ID–SUP:

$$\frac{\phi_2^1 \wedge \phi_2^2 \Rightarrow \phi_2 \qquad \textit{(premise)}}{\{\alpha\Delta^1, \alpha\Delta^2\} \Rightarrow \alpha\Delta \quad \textit{(conclusion)}}$$

Proof of rule 5.25. By lemma 5.2 (characterizing formula $\phi_{\alpha\Delta}$ of LSTD–diagram $\alpha\Delta$),

$$\phi_{\alpha\Delta} \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{next}\, \phi_{\Delta_1}) \quad ;$$

the formulas $\phi_{\alpha\Delta^1}$and $\phi_{\alpha\Delta^2}$are defined similarly, where ϕ_2^1 respectively ϕ_2^2 replaces ϕ_2. By premise,

$$\begin{array}{lll} \phi_{\alpha\Delta^1} \wedge \phi_{\alpha\Delta^2} & \sim & (\phi_2^1 \wedge \phi_2^2) \vee (\phi_1 \wedge \mathbf{next}\, \phi_{\Delta_1}) \\ \text{[by premise]} & \Rightarrow & \phi_{\alpha\Delta} \end{array}$$

Rule 5.26 (LSTD–invariant–diagram superposition) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formula ϕ_1, and LSTD–bodies Δ_1^1, Δ_1^2 and Δ_1 over V.*

Let

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!-} \Delta_1^1 \quad ,$$

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!-} \Delta_1^2$$

and

$$\alpha\Delta \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!-} \Delta_1 \quad .$$

Then the following rule holds:

Rule LSTD–AD–SUP:

$$\frac{\{\Delta_1^1, \Delta_1^2\} \Rightarrow \Delta_1 \qquad \textit{(premise)}}{\{\alpha\Delta^1, \alpha\Delta^2\} \Rightarrow \alpha\Delta \qquad \textit{(conclusion)}}$$

Proof of rule 5.26. By lemma 5.2 (characterizing formula $\phi_{\alpha\Delta}$ of LSTD–diagram $\alpha\Delta$),

$$\phi_{\alpha\Delta} \equiv \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,\phi_{\Delta_1}) \quad ;$$

the formulas $\phi_{\alpha\Delta^1}$and $\phi_{\alpha\Delta^2}$are defined similarly, where $\phi_{\Delta_1^1}$ respectively $\phi_{\Delta_1^2}$ replaces ϕ_{Δ_1}.

The following derivation establishes the conclusion:

$$\begin{array}{rcc}
\phi_{\alpha\Delta^1} \wedge \phi_{\alpha\Delta^2} & \sim & \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,\phi_{\Delta_1^1}) \wedge \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,\phi_{\Delta_1^2}) \\
\text{[by LTL–tautology]} & \sim & \textbf{always}\,((\phi_1 \rightarrow \textbf{next}\,\phi_{\Delta_1^1}) \wedge (\phi_1 \rightarrow \textbf{next}\,\phi_{\Delta_1^2})) \\
\text{[by prop–tautology]} & \sim & \textbf{always}\,(\phi_1 \rightarrow (\textbf{next}\,\phi_{\Delta_1^1} \wedge \textbf{next}\,\phi_{\Delta_1^2})) \\
\text{[by LTL–tautology]} & \sim & \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,(\phi_{\Delta_1^1} \wedge \phi_{\Delta_1^2})) \\
\text{[by premise]} & \Rightarrow & \textbf{always}\,(\phi_1 \rightarrow \textbf{next}\,(\phi_{\Delta_1})) \\
 & \sim & \phi_{\alpha\Delta}
\end{array}$$

5.4.4 Transformation of LSTD–specifications

Recall that by definition 5.5 a LSTD–specification (LSTD–Spec) ΔS is a finite set of LSTD–diagrams over V, i.e. for some $k \geq 1$

$$\Delta S = \{\alpha\Delta_i \mid i = 1 \ldots k\} \quad .$$

The semantics of a LSTD–Spec ΔS, denoted by $L(\Delta S)$, is defined to be the intersection of the semantics of the diagrams contained in ΔS, and can by lemma 5.3 be characterized by the temporal logic formula

$$\phi_{\Delta SPEC} =_{Def} \bigwedge_{i=1\ldots k} \phi_{\alpha\Delta_i} \quad .$$

We will next formulate rules with conclusions of the form

Rule ⟨name⟩:

$$\frac{\ldots}{\{\alpha\Delta_1, \ldots, \alpha\Delta_k\} \Rightarrow \{\alpha\Delta'_1, \ldots, \alpha\Delta'_r\}} \quad \begin{array}{l}\text{(premise)} \\ \text{(conclusion)}\end{array}$$

which means that the premise ... implies

$$\phi_{\Delta SPEC} \Rightarrow \phi_{\Delta SPEC'}$$

where

$$\Delta S \equiv_{Def} \{\alpha\Delta_1, \ldots, \alpha\Delta_k\}$$

and

$$\Delta S' \equiv_{Def} \{\alpha\Delta'_1, \ldots, \alpha\Delta'_r\} \quad .$$

The following transformations can be applied to LSTD–specifications:

- diagrams can be omitted from the set;
- diagrams can be replaced by similar diagrams, according to on of the weakening rules presented in this chapter;
- further diagrams can be adjoined to the set, which follow by logical implication (tautology) from the diagrams in the set. The implication can be established e.g. by the superposition rule, or by some other method (in particular tautology checking).

Rule 5.27 (LSTD–specification diagram elimination) *Assume an assertion language $A\mathcal{L}$, a set V of variables , and an LSTD–specification*

$$\Delta S = \{\alpha\Delta_i \mid i = 1 \ldots k\}$$

over V (for some $k \geq 1$). Let $\alpha\Delta \in \Delta S$ be a diagram in the set ΔS.
Then the following rule holds (without premise):

Rule LSTD–SPW–1:

$$\frac{}{\Delta S \Rightarrow \Delta S \backslash \{\alpha\Delta\}} \qquad \textit{(conclusion)}$$

The proof of rule 5.27 goes without saying.

Rule 5.28 (LSTD–specification diagram weakening) *Assume an assertion language $A\mathcal{L}$, a set V of variables , and an LSTD–specification*

$$\Delta S = \{\alpha\Delta_i \mid i = 1 \ldots k\}$$

over V (for some $k \geq 1$). Let $\alpha\Delta \in \Delta S$ be a diagram in the set ΔS, and $\alpha\Delta'$ be a similar LSTD–diagram.

Then the following rule holds:

Rule LSTD–SPW–2:

$$\frac{\alpha\Delta \Rightarrow \alpha\Delta'}{\Delta S \Rightarrow \Delta S \cup \{\alpha\Delta'\}} \quad \begin{array}{l} \textit{(premise)} \\ \textit{(conclusion)} \end{array}$$

The proof of rule 5.28 is obvious: From the premise,

$$\phi_{\alpha\Delta} \Rightarrow \phi_{\alpha\Delta'}$$

and $\alpha\Delta \equiv \alpha\Delta_j$ for some $j, 1 \leq j \leq k$, so

$$\phi_{\Delta SPEC} =_{Def} \bigwedge_{i=1\ldots k} \phi_{\alpha\Delta_i}$$

implies

$$\phi_{\Delta SPEC \cup \{\alpha\Delta'\}} =_{Def} \bigwedge_{i=1\ldots k} \phi_{\alpha\Delta_i} \wedge \phi_{\alpha\Delta'} \quad .$$

Rule 5.28 can be applied if one of the LSTD–diagram weakening rules can be used to establish the premise.

A general form of this rule is given next.

Rule 5.29 (LSTD–specification diagram implication) *Assume an assertion language $A\mathcal{L}$, a set V of variables , and an LSTD–specification*

$$\Delta S = \{\alpha\Delta_i \mid i = 1 \ldots k\}$$

over V (for some $k \geq 1$). Let $\Delta S' \subseteq \Delta S$ be a subset of the diagram in the set ΔS, and $\alpha\Delta'$ be a similar LSTD–diagram.

Then the following rule holds:

Rule LSTD–SPW–3:

$$\frac{\Delta S' \Rightarrow \alpha\Delta' \qquad \textit{(premise)}}{\Delta S \Rightarrow \Delta S \cup \{\alpha\Delta'\} \qquad \textit{(conclusion)}}$$

The proof of this rule is omitted; it is similar to the proof of rule 5.28.

This rule is the basis for a verification environment which uses tautology checking. New diagrams are derived from an existing set of (established) diagrams by choosing a subset $\Delta S'$ of the existing set as premise, and the new diagram $\alpha\Delta'$ as "implication goal".

The premise is established by checking that the temporal logic formula

$$\phi_{\Delta SPEC'} \rightarrow \phi_{\alpha\Delta'}$$

is a tautology.

The rule is applicable whenever tautology–checking is possible, which requires that the assertion language (at least the fragment relevant for the proof) contains only finite data–types. Note that practical experience shows that complexity problems (known to be the major obstacle of model–checking) seldom arise with this approach, since the set of diagrams used in the premise of rule 5.28 is considerably small.

5.5 Compositional reasoning

We reconsider again the distributed version of the Req/Ack–protocol introduced in example Now, reconsider the *Slave*–module of the *distributed* version of the Req/Ack–protocol introduced in example 4.2.

The slave–module was defined as

$$\begin{array}{rc} \mathcal{G}M_{Ack} : & \textbf{module} \text{ Slave} \\ & \textbf{out} \text{ Ack : Bit } \textbf{where} \text{ Ack = '0'} \\ & \textbf{external} \text{ Req: Bit} \\ & \mathcal{G}_2 \end{array}$$

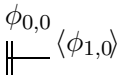
where $\mathcal{G}_2$ is defined as shown in figure 5.1.

We have argued that a "local" specification of the Slave–module with respect to the interface variables Req, Ack is

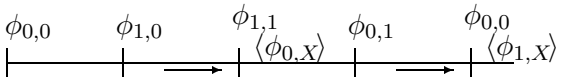
$$\Delta S^S \equiv_{Def} \{\alpha\Delta_0^S, \alpha\Delta_1^S\}$$

where

Initialization:

$\alpha\Delta_0^S \equiv_{Def}$ 

Invariance:

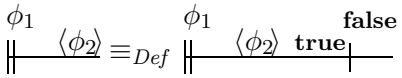
$\alpha\Delta_1^S \equiv_{Def}$ 

and

$$\phi_{x,y} \equiv_{Def} \langle\!|Req = \mathrm{x}|\!\rangle \wedge \langle\!|Ack = \mathrm{y}|\!\rangle \quad ,$$

using the convention that $\langle\!|\langle \mathrm{sig}\rangle = X|\!\rangle \equiv_{Def} \mathbf{true}$.

For reasons of uniform presentation, an initial diagram is required by definition 5.2 to have a non–empty body. We define a "pure" (static) initialization to be an abbreviation as follows:



Let

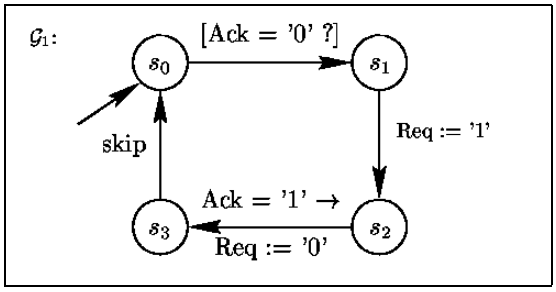
$$\alpha\Delta \equiv_{Def} \phi_1 \Vdash \langle\phi_2\rangle$$

Then the semantics of $\alpha\Delta$ is by definition of the abbreviation:

$$\begin{array}{lll} \phi_{\alpha\Delta} & \equiv_{Def} & \phi_2 \vee (\phi_1 \wedge \\ & \sim & \mathbf{next}\,(\mathbf{true}\,\mathbf{unless}\,\mathbf{false})) \\ & \sim & \phi_2 \vee \phi_1 \quad . \end{array}$$

The master–module is defined similar to the slave–module:

Figure 5.2: Implementation of Master–model of the basic 4–phase handshake protocol.



$$\mathcal{G}M_{Req} \; : \quad \textbf{module} \;\; \text{Master}$$
$$\textbf{out} \;\; \text{Req} : \text{Bit} \;\; \textbf{where} \;\; \text{Req} = \text{'0'}$$
$$\textbf{external} \;\; \text{Ack: Bit}$$
$$\mathcal{G}_1$$

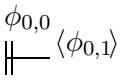
where $\mathcal{G}_1$ is defined as shown in figure 5.2.

We claim that a "local" specification of the Master–module with respect to the interface variables Req, Ack is

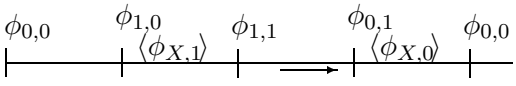
$$\Delta S^M \equiv_{Def} \{\alpha\Delta_0^M, \alpha\Delta_1^M\}$$

where

Initialization:

$\alpha\Delta_0^M \equiv_{Def}$ 

Invariance:

$\alpha\Delta_1^M \equiv_{Def}$ 

and

$$\phi_{x,y} \equiv_{Def} \langle\!|Req =\mathrm{x}|\!\rangle \wedge \langle\!|Ack =\mathrm{y}|\!\rangle \quad ,$$

We will next show that the conjunctive effect of the local specification of the Slave and the local specification of the Master implies the specification of the *Req/Ack*–protocol, defined as follows:

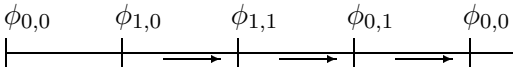
$$\Delta S^{M\|S} \equiv_{Def} \{\alpha\Delta_0, \alpha\Delta_1\}$$

where

Initialization:

$\alpha\Delta_0 \equiv_{Def}$ 

Invariance:

$\alpha\Delta_1 \equiv_{Def}$ 

We next consider a reformulation of the composition rule R–comp formulated in chapter 4.

Rule 5.30 (LSTD–composition rule) *Let $\mathcal{G}M_1$, $\mathcal{G}M_2$ be OTGS, $\mathcal{G}M$ a composition of $\mathcal{G}M_1$ and $\mathcal{G}M_2$, and ΔS^1,ΔS^2 LSTD–specifications over the set of variables defined by union of the local interfaces of $\mathcal{G}M_1$ respectively $\mathcal{G}M_2$.*

Then the following rule holds :

Rule LSTD–COMP:

$$\frac{\begin{array}{c}\mathcal{G}M \ : \ \mathcal{G}M_1\|\mathcal{G}M_2 \\ \mathcal{G}M_1 \models \Delta S^1 \quad , \quad \mathcal{G}M_2 \models \Delta S^2\end{array}}{\mathcal{G}M \models \Delta S^1 \cup \Delta S^2}$$

This rule is a direct consequence of theorem 4.1, where $\phi_1 \sim \phi_{\Delta SPEC^1}$, $\phi_2 \sim \phi_{\Delta SPEC^2}$ and

$$\phi_1 \wedge \phi_2 \sim \phi_{\Delta SPEC^1 \cup \Delta SPEC^2} \quad .$$

The next rule complements the basis of compositional reasoning.

Rule 5.31 (LSTD–derivation rule) *Let $\mathcal{G}M$ be an OTGS, and ΔS^1,ΔS^2 LSTD–specifications over the set of variables defined by the interface of $\mathcal{G}M$.*

Then the following rule holds :

Rule LSTD–DERIV:

$$\frac{\begin{array}{l}\mathcal{G}M \models \Delta S^1 \\ \Delta S^1 \Rightarrow \Delta S^2\end{array}}{\mathcal{G}M \models \Delta S^2}$$

The proof of this rule is an obvious consequence of the transitivity of language inclusion:

$$L(\mathcal{G}M) \subseteq L(\Delta S^1) \subseteq L(\Delta S^2)$$

We will demonstrate the application of proof rules in the next example. As preparation, we need a few abbreviations.

Let

$$\Delta_4^S \equiv_{Def} \overset{\phi_{0,0}}{\underset{\phi_{0,1}}{\Longrightarrow\!\!\mid}} \langle\phi_{1,X}\rangle$$

$$\Delta_3^S \equiv_{Def} \overset{\phi_{0,1}}{\underset{\phi_{1,1}}{\mid}} \Delta_4^S$$

$$\Delta_2^S \equiv_{Def} \overset{\phi_{1,1}}{\underset{\phi_{1,0}}{\rightarrow\!\!\mid}} \langle\phi_{0,X}\rangle \Delta_3^S$$

$$\Delta_1^S \equiv_{Def} \overset{\phi_{1,0}}{\underset{\phi_{0,0}}{\mid}} \Delta_2^S$$

Similary, let

Figure 5.3: Basis of derivation of the protocol of the distributed Master/Slave–system.

$$\text{[Rule 5.30]}\ \frac{\text{[Rule MC]}\ \overline{\mathcal{G}M_{Req} \models \Delta S^M} \quad , \quad \text{[Rule MC]}\ \overline{\mathcal{G}M_{Ack} \models \Delta S^S}}{\mathcal{G}M_{Req,Ack} \models \Delta S^M \cup \Delta S^S}$$

$$\begin{aligned}
\Delta_4^M &\equiv_{Def} \overset{\phi_{0,0}}{\underset{\phi_{0,1}}{\vdash}} \\
\Delta_3^M &\equiv_{Def} \overset{\phi_{0,1}}{\underset{\phi_{1,1}}{\Longrightarrow\!\vdash}} \langle\phi_{X,0}\rangle\ \Delta_4^M \\
\Delta_2^M &\equiv_{Def} \overset{\phi_{1,1}}{\underset{\phi_{1,0}}{\vdash}} \Delta_3^M \\
\Delta_1^M &\equiv_{Def} \overset{\phi_{1,0}}{\underset{\phi_{0,0}}{\vdash}} \langle\phi_{X,1}\rangle\ \Delta_2^M
\end{aligned}$$

Together with abbreviation introduced earlier, we get

$$\alpha\Delta_1^S \equiv \overset{\phi_{0,0}}{\vdash\!\!\!-} \Delta_1^S$$

and

$$\alpha\Delta_1^M \equiv \overset{\phi_{0,0}}{\vdash\!\!\!-} \Delta_1^M \quad .$$

Example 5.4 (Derivation of basic 4–phase handshake protocol) *The following sequence of figures shows a derivation of the basic 4–phase handshake protocol .*

The figure 5.3 shows the basis of the derivation of the protocol of the combined system. The LSTD–specifications of the Master– and the Slave–module are assumed to be verified by model–checking. This means that we take the validity of the local specifications for granted.

Next, the composition rule is applied, which allows to continue reasoning on a purely logical level (i.e., without taking the semantics of module composition into further account).

First, we derive the initial diagram $\alpha\Delta_0$ from the corresponding local spec-

Figure 5.4: Derivation of initialization of the basic 4–phase–handshake protocol.

$$\dfrac{\dfrac{}{\phi_{1,0} \wedge \phi_{0,1} \Rightarrow \textbf{false}}}{\dfrac{}{\{\alpha\Delta_0^M, \alpha\Delta_0^S\} \Rightarrow \underbrace{\overset{\phi_{0,0}}{\Vdash\!\!\!-\!\!\!-}}_{\alpha\Delta_0}}\ [\text{Rule 5.25}]}{\Delta S^M \cup \Delta S^S \Rightarrow \Delta S^M \cup \Delta S^S \cup \{\alpha\Delta_0\}}\ [\text{Rule 5.29}]$$

ifications $\alpha\Delta_0^M \in \Delta S^M$ *and* $\alpha\Delta_0^S \in \Delta S^S$.

Second, we derive the invariant diagram $\alpha\Delta_1$ *from the corresponding local specifications* $\alpha\Delta_1^M \in \Delta S^M$ *and* $\alpha\Delta_1^S \in \Delta S^S$. *The derivation tree is constructed by repeated application of rule 5.9 and 5.18.*

5.6 Summary

The approach of LSTD presented in this chapter has some similarity with the UNITY programming logic [9]. In particular, the idea of compositional reasoning by specification composition, which has been illustrated for the UNITY–programming notation by many examples in the book [9], has motivated the structure of LSTD–specifications.

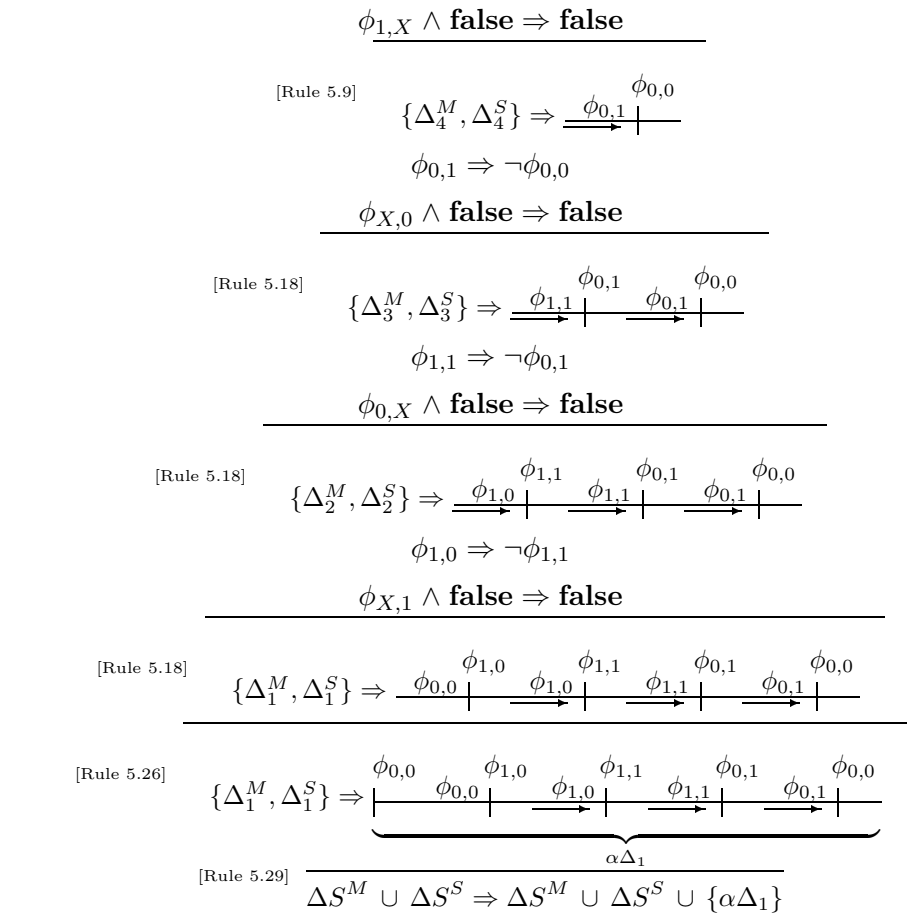
The availability of proof rules for LSTD opens the way for an axiomatization of the proof rules in an interactive theorem proving environment. A prototype implementation of a LSTD–theory for LSTD, working in cooperation with an axiomatization of a subset of VHDL–expression language, has been developed on basis of the LAMBDA–theorem prover. The results of this project have been surveyed in [30].

The set of proof rules presented in this chapter is not complete. Many more proof rules can be formulated; in particular, a proof rule called *chaining* will be introduced in the next chapter.

On the other hand, given that we consider a verification methodology based on a combination of model–checking and tautology–checking, all datatypes can be assumed to have finite–data types (although this requirement can be relaxed by the application of abstraction techniques). Thus, rule 5.29 alone would suffice to ensure completeness of the derivation rules.

Proofs of the sort shown in example 5.4 are hard to find in practice. Often, the interplay of diagrams is very involved, and can only be decided by a tableaux–analysis.

Figure 5.5: Derivation of invariant part of the basic 4–phase–handshake protocol.



The idea of *semantical analysis* of a LSTD–specification can be taken further towards an approach of *automatic synthesis from requirements specifications*. A realization of this approach has been implemented early along with the development of STD in the ICOS system [23], and has been recently extended, now constituting the component of a rapid–prototyping–environment ([25]).

Chapter 6

Symbolic Timing Diagrams

The concept of LSTD–diagrams, which has been introduced in chapter 5, is based on a mathematical notation. In particular, the two–dimensional "visual" characteristic, which is typical of conventional timing diagrams, has not been introduced yet.

In fact, LSTD–diagrams could be considered as a visualization of a sub–dialect of (linear) temporal logic, which has certain useful mathematical properties supporting compositional reasoning.

In this chapter we will define and analyze the formalism of Symbolic Timing Diagrams (STD), which has already been introduced informally in chapter 2.

The STD formalism has more syntactical elements than the LSTD formalism, in particular the syntactic category of *constraints*, which serve to express restrictions on the relative occurrence sequence of events on two different waveforms.

However, this additional notation merely adds to the convenience of notation and to the graphical appeal. As a main result of this chapter, it will be shown that an STD–specification can be translated into an equivalent LSTD–specification.

6.1 LSTD–diagram composition

Before the introduction of STD, we will further extend the set of rules for transformation of LSTD–diagrams.

In particular, we will consider a composition operation for LSTD–diagrams called *chaining*, which consists of the extension of one LSTD–diagram body by another body of a "matching" invariant LSTD–diagram.

6.1.1 Chaining

The idea of chaining is derived from the observation, that an valid LSTD diagram with "invariant" activation mode can be interpreted as a (dynamic) *invariant property.*

This is expressed precisely by the next rule.

Rule 6.1 (LSTD–chaining) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas ϕ_1 and $\phi_1^i, \phi_2^i, \phi_3^i, i = 1 \ldots r$, and an LSTD–bodies Δ, Δ_1 over V,*

where

$$\Delta \equiv_{Def} \Delta E_1 \ldots \Delta E_r \quad ,$$

$$\Delta E_i \equiv_{Def} \overset{\phi_2^i}{\underset{\phi_1^i}{\dashv}}\langle\phi_3^i\rangle \quad or \quad \Delta E_i \equiv_{Def} \overset{\phi_2^i}{\underset{\phi_1^i}{\Longrightarrow\!\dashv}}\langle\phi_3^i\rangle \quad , \; i = 1 \ldots r \; ,$$

and

$$\alpha\Delta \equiv_{Def} \overset{\phi_1}{\vdash} \Delta_1 \quad .$$

Then the following rule holds:

Rule LSTD–CHAIN:

$$\frac{\phi_2^r \Rightarrow \phi_1}{\{\Delta, \alpha\Delta\} \Rightarrow \Delta \; \Delta_1} \qquad \begin{matrix} \textit{(premise)} \\ \textit{(conclusion)} \end{matrix}$$

Proof of rule 6.1. We have to show that

$$\phi_\Delta \wedge \phi_{\alpha\Delta} \Rightarrow \phi_{\Delta\Delta_1} \quad ,$$

i.e.

$$\forall\sigma \ : \ \sigma \models \phi_\Delta \wedge \phi_{\alpha\Delta} \rightarrow \phi_{\Delta\Delta_1} \quad .$$

In an equivalent formulation,

$$\forall\sigma \ : \ \sigma \models \phi_{\alpha\Delta} \rightarrow (\phi_\Delta \rightarrow \phi_{\Delta\Delta_1}) \quad ,$$

which is the same as

$$\phi_{\alpha\Delta} \Rightarrow (\phi_\Delta \rightarrow \phi_{\Delta\Delta_1}) \quad . \qquad (*)$$

Formula ϕ_Δ can be written in the form:

$$\phi_\Delta \ =_{Def} \ \Phi[\phi_1/u]$$

where

$$\begin{array}{rlll} \Phi \equiv_{Def} & (\phi_1^1 \ op^1 & ((\phi_3^1 \wedge \neg\phi_1^1 \wedge \neg\phi_2^1) \vee \\ & & (\phi_2^1 \wedge \textbf{next} \\ & (\phi_1^2 \ op^2 & ((\phi_3^2 \wedge \neg\phi_1^2 \wedge \neg\phi_2^2) \vee \\ & & (\phi_2^2 \wedge \textbf{next} \\ & \ldots & \\ & (\phi_1^r \ op^r & ((\phi_3^r \wedge \neg\phi_1^r \wedge \neg\phi_2^r) \vee \\ [+] & & (\phi_2^r \wedge u) \quad) \,)) \,) \,)) \ldots)) \end{array}$$

and op^i is either the operator **unless** or **until**, dependent on the definition of ΔE_i. Note that the extension marked [+] ist justified by the premise, which implies that

$$\phi_2^r \wedge \phi_1 \sim \phi_2^r \quad .$$

From the definition of the construction of the characterizing formula of the LSTD–body $\Delta\Delta_1$ it is easy to see that

$$\phi_{\Delta\Delta_1} \equiv \Phi[\mathbf{next}\,\phi_{\Delta_1}/u] \quad .$$

From lemma 3.8 we know that

$$\mathbf{always}\,(\phi_1 \rightarrow \mathbf{next}\,\phi_{\Delta_1}) \Rightarrow \Phi[\phi_1/u] \rightarrow \Phi[\mathbf{next}\,\phi_{\Delta_1}/u] \quad .$$

This is equivalent to (*), since by definition

$$\phi_{\alpha\Delta} = \mathbf{always}\,(\phi \rightarrow \mathbf{next}\,\phi_{\Delta_1}) \quad ;$$

hence rule 6.1 is proven.

An immediate and useful consequence of the chaining rule is the following rule about LSTD *extension*.

Rule 6.2 (LSTD–extension) *Assume the same assumption as for rule 6.1, in particular*

$$\Delta \equiv_{Def} \Delta E_1 \ldots \Delta E_r \quad ,$$

and some LSTD–body Δ_1.

Then the following rule holds:

Rule LSTD–EXT:

$$\frac{\phi_{\Delta_1} \sim \mathbf{true}}{\Delta \Rightarrow \Delta\ \Delta_1} \qquad \begin{array}{l}(premise)\\(conclusion)\end{array}$$

Proof of rule 6.2. First, note that $\mathbf{next}\,\mathbf{true} \sim \mathbf{true}$. Using the same notation as in the proof of rule 6.1, we get:

$$\phi_{\Delta\Delta_1} = \Phi[\mathbf{next}\,\phi_{\Delta_1}/u] \sim \Phi[\mathbf{true}/u] \sim \phi_{\Delta} \quad ,$$

so $\phi_{\Delta\Delta_1} \sim \phi_{\Delta}$, hence the conclusion holds in particular.

Example 6.1 (Complementary event extension) *Assume an assertion language* $A\mathcal{L}$*, a set* V *of variables , and Boolean formula* ϕ *. Let*

$$\Delta \equiv_{Def} \Delta E_1 \ldots \Delta E_r \quad ,$$

be an LSTD–body over V and

$$\Delta E \equiv_{Def} \overset{\phi}{\underset{\neg\phi}{\vdash\!\!\!-\!\!\!-}} \quad .$$

Then

$$\Delta \Rightarrow \Delta\ \Delta E \quad .$$

This follows from rule 6.2, because

$$\phi_{\Delta E} \sim \neg\phi\ \mathbf{unless}\ \phi \sim \mathbf{true} \quad .$$

The next rule allows to apply rule 6.1 on the level of LSTD–specifications.

Rule 6.3 (LSTD–diagram–chaining) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas ϕ_1 and $\phi_1^i, \phi_2^i, \phi_3^i, i = 1 \ldots r$, and LSTD–bodies Δ, Δ_1 over V, where*

$$\Delta \equiv_{Def} \Delta E_1 \ldots \Delta E_r \quad ,$$

$$\Delta E_i \equiv_{Def} \overset{\phi_2^i}{\underset{\phi_1^i}{\vdash\!\!\!-\!\!\!-}}\ \langle\phi_3^i\rangle \quad or \quad \Delta E_i \equiv_{Def} \overset{\phi_2^i}{\underset{\phi_1^i}{\Longrightarrow\!\!\!\!\vdash}}\ \langle\phi_3^i\rangle \quad , \ i = 1 \ldots r \ ,$$

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi'}{\vdash\!\!\!-} \Delta \quad ,$$

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!-} \Delta_1 \quad ,$$

and

$$\alpha\Delta \equiv_{Def} \overset{\phi'}{\vdash\!\!\!-} \Delta\ \Delta_1 \quad .$$

Then the following rule holds:

Rule LSTD–AD–CHAIN:

$$\frac{\phi_2^r \Rightarrow \phi_1}{\{\alpha\Delta^1, \alpha\Delta^2\} \Rightarrow \alpha\Delta} \qquad \begin{array}{l} \textit{(premise)} \\ \textit{(conclusion)} \end{array}$$

Proof of rule 6.3. The proof is completely analogue to the proof of rule 6.1.

We have to show that

$$\phi_{\alpha\Delta^1} \wedge \phi_{\alpha\Delta^2} \Rightarrow \phi_{\alpha\Delta} \quad ,$$

which is the same as

$$\phi_{\alpha\Delta^2} \Rightarrow (\phi_{\alpha\Delta^1} \rightarrow \phi_{\alpha\Delta}) \quad . \qquad (*)$$

Formula $\phi_{\alpha\Delta^1}$ can be written in the form:

$$\phi_{\alpha\Delta^1} =_{Def} \Phi^{\alpha}[\phi_1/u]$$

where

$$\Phi^{\alpha} \equiv_{Def} \mathbf{always}\,(\phi' \rightarrow \mathbf{next}\,\Phi)$$

and Φ is defined as in the proof of rule 6.1.

It is easy to see that

$$\phi_{\alpha\Delta} \equiv \Phi^{\alpha}[\mathbf{next}\,\phi_{\Delta_1}/u] \quad .$$

By lemma 3.8 ,

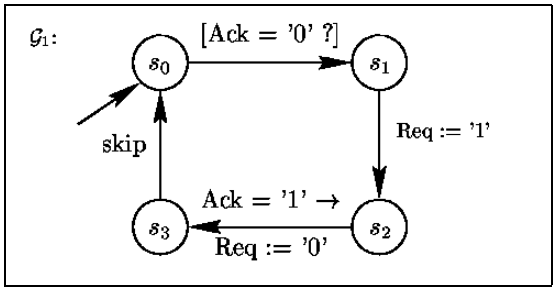
$$\mathbf{always}\,(\phi_1 \rightarrow \mathbf{next}\,\phi_{\Delta_1}) \Rightarrow \Phi^{\alpha}[\phi_1/u] \rightarrow \Phi^{\alpha}[\mathbf{next}\,\phi_{\Delta_1}/u]$$

which is equivalent to (*), since

$$\phi_{\alpha\Delta^2} = \mathbf{always}\,(\phi_1 \rightarrow \mathbf{next}\,\phi_{\Delta_1}) \quad ;$$

hence rule 6.3 is proven.

Figure 6.1: Implementation of Master–model of the basic 4–phase handshake protocol.



We reconsider as example the local specification of the master component of the basic 4–phase handshake protocol.

Example 6.2 (Phase–level specification of master–protocol) *Recall the definition of the master–module introduced in chapter 5 :*

$$\begin{array}{rc} \mathcal{G}M_{Req} : & \textbf{module} \ \text{Master} \\ & \textbf{out} \ \text{Req : Bit } \textbf{where} \ \text{Req = '0'} \\ & \textbf{external} \ \text{Ack: Bit} \\ & \mathcal{G}_1 \end{array}$$

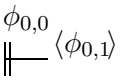
where $\mathcal{G}_1$ is defined as shown in figure 6.1.

We claim that the implementation shown in figure 6.1 satisfies the following local specification with respect to the interface variables Req, Ack :

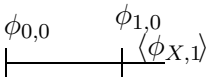
$$\Delta S_4^M \equiv_{Def} \{\alpha\Delta_0^M, \alpha\Delta_{11}^M, \alpha\Delta_{12}^M, \alpha\Delta_{13}^M, \alpha\Delta_{14}^M\}$$

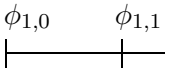
where

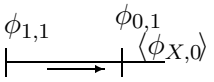
Initialization:

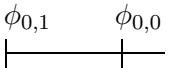
$\alpha\Delta_0^M \equiv_{Def}$ 

Invariance:

$\alpha\Delta_{11}^M \equiv_{Def}$ 

$\alpha\Delta_{12}^M \equiv_{Def}$ 

$\alpha\Delta_{13}^M \equiv_{Def}$ 

$\alpha\Delta_{14}^M \equiv_{Def}$ 

and

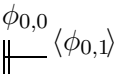
$$\phi_{x,y} \equiv_{Def} \langle\!|Req = x|\!\rangle \wedge \langle\!|Ack = y|\!\rangle \quad .$$

We will next show that the specification of the Master–component defined in example 6.2 implies the specification of the master–component introduced in chapter 5 as basis of the protocol derivation shown in example 5.4:

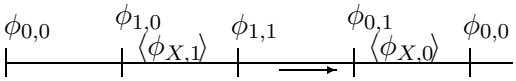
$$\Delta S^M \equiv_{Def} \{\alpha\Delta_0^M, \alpha\Delta_1^M\}$$

where

Initialization:

$\alpha\Delta_0^M \equiv_{Def}$ 

Invariance:

$\alpha\Delta_1^M \equiv_{Def}$ 

and

$$\phi_{x,y} \equiv_{Def} \langle\!|Req = \mathrm{x}|\!\rangle \wedge \langle\!|Ack = \mathrm{y}|\!\rangle \quad .$$

Example 6.3 (Local specification derivation) *Let* ΔS_4^M *and* ΔS^M *be specifications of the master component, as defined previously .*

Then

$$\frac{}{\Delta S_4^M \Rightarrow \Delta S^M} \qquad (*)$$

By definition,

$$\Delta S^M = \{\alpha\Delta_0^M, \alpha\Delta_1^M\} \quad ;$$

since $\alpha\Delta_0^M \in \Delta S_4^M$, it suffices to prove that

$$\Delta S_4^M \Rightarrow \alpha\Delta_1^M \quad .$$

This is demonstrated by the derivation shown in figure 6.2, where the following abbreviations are used:

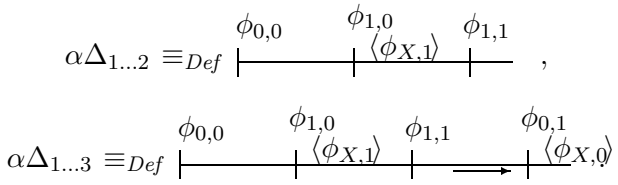
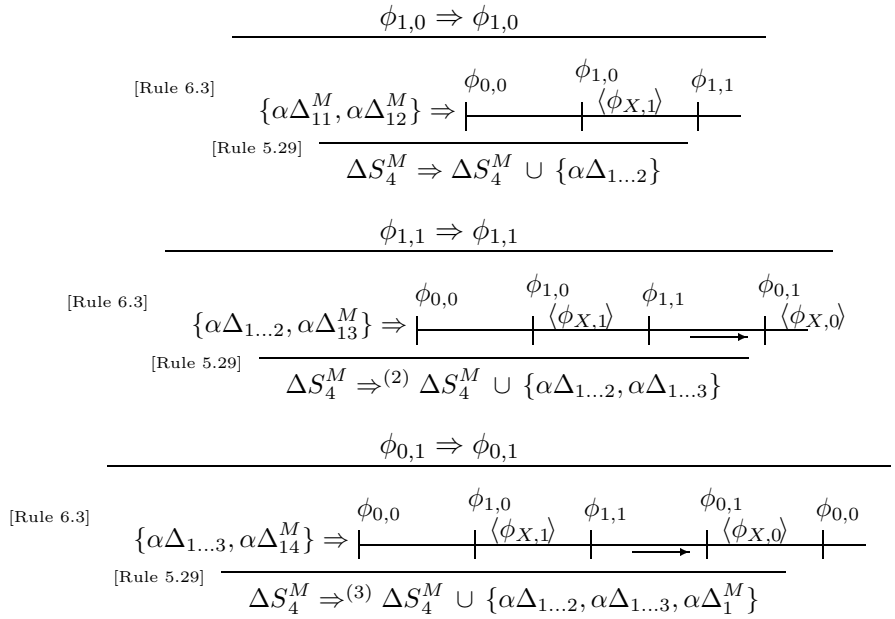


Figure 6.2: Derivation of the LSTD–diagram $\alpha\Delta_1^M$.



6.1.2 Parallel composition

The operation of diagram–chaining can be described, figuratively speaking, as "horizontal" composition.

It is also possible to define a "vertical" composition operation, which will lead us to the definition of STD–diagrams.

First, consider the following simple construction: Given two LSTD–diagrams $\alpha\Delta^1, \alpha\Delta^2$, defined by

$$\alpha\Delta^1 \equiv_{Def} \overset{\phi_1}{\vdash\!\!\!-\!\!\!-} \Delta_1^1$$

and

$$\alpha\Delta^2 \equiv_{Def} \overset{\phi_1'}{\vdash\!\!\!-\!\!\!-} \Delta_1^2 \quad ,$$

we could build the following diagram:

$$\binom{\alpha\Delta^1}{\alpha\Delta^2} \equiv_{Def} \begin{array}{l} \overset{\phi_1 \wedge \phi_1'}{\vdash\!\!\!-\!\!\!-} \Delta_1^1 \\ \vdash\!\!\!-\!\!\!-\ \Delta_1^2 \end{array}$$

The construction $\binom{\alpha\Delta^1}{\alpha\Delta^2}$ is called the *parallel composition* of the diagrams $\alpha\Delta^1$ and $\alpha\Delta^2$.

We define the semantics of $\binom{\alpha\Delta^1}{\alpha\Delta^2}$ as equivalent to the semantics of the LSTD–specification $\Delta S_{1\|2}$:

$$L(\binom{\alpha\Delta^1}{\alpha\Delta^2}) \equiv_{Def} L(\Delta S_{1\|2})$$

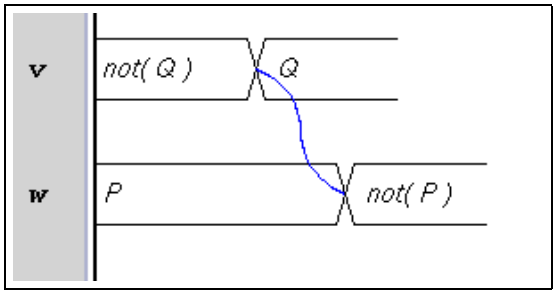
where

$$\Delta S_{1\|2} \equiv_{Def} \{ \quad \overset{\phi_1 \wedge \phi'}{\vdash\!\!\!-\!\!\!-} \Delta_1^1 \quad , \quad \overset{\phi_1 \wedge \phi_1'}{\vdash\!\!\!-\!\!\!-} \Delta_1^2 \quad \}$$

i.e. the activation condition of the parallel composition is distributed over the waveforms occuring in the composition.

There are no new effects introduced by the pure parallel composition of LSTD–diagrams. The semantics allows all possible interleavings of the occurrence of events on the waveforms of a parallel composition.

Figure 6.3: STD Diagram *TL_P_and_not_Q_unless_Q*.



The classical notion of timing diagrams uses *timing constraints* to express restrictions on the occurrence time of different events of a diagram. Similary, STD uses constraints to define (primarily) restrictions on the *order of the occurrence* of events. By definition, the events on a waveform are linearly ordered. Thus, ordering constraints relate events on different waveforms.

Some examples have been introduced earlier in chapter 2. For instance, reconsider the diagram shown in figure 6.3.

The diagram named *TL_P_and_not_Q_unless_Q* can be considered to be a parallel composition of the LSTD–diagrams

$$\alpha\Delta^1 \equiv_{Def} \quad \langle\!|not(Q)|\!\rangle \quad e_1 : \langle\!|Q|\!\rangle$$

and $$\alpha\Delta^2 \equiv_{Def} \quad \langle\!|P|\!\rangle \quad e_2 : \langle\!|not(P)|\!\rangle$$

with an additional ordering constraint $e_1 \rightarrow e_2$, which requires that event e_2 must not occur before event e_1.

STD–diagram can thus be considered to be a parallel composition of two or more LSTD–diagrams constituting the waveforms of an STD–diagram, possibly equipped with additional constraints between the events of the waveforms.

6.2 Structure of STD–diagrams

The next definition introduces the concept of an STD–diagram as parallel composition of LSTD diagrams with constraints. The definition is more elaborate than the definition of LSTD–diagrams, since it is oriented to the concrete graphical representation of STD–diagrams.

6.2.1 Definition of STD–diagrams

An STD–diagram contains a named collections of LSTD–diagrams with equal activation mode (initial or invariant). The names are drawn from a designated set ID^W of *waveform names*.

Definition 6.1 (STD–diagram) *Assume an assertion language $A\mathcal{L}$, a set V of variables , Boolean formulas ϕ^1_w and LSTD–bodies Δ_w over V (the index w is specified below).*

We define a "bundle" $\mathcal{W}$ of waveforms over V to be a collection of LSTD–diagrams with equal activation mode:

$$\mathcal{W} = (w \in ID_{\mathcal{W}} \mapsto \alpha\Delta_w)$$

where $ID_{\mathcal{W}} \subseteq ID^W$ is a finite set of diagram waveform names and either

$$\alpha\Delta_w \equiv_{Def} \overset{\phi^1_w}{\longmapsto} \Delta_w$$

$$or \quad \alpha\Delta_w \equiv_{Def} \overset{\phi^1_w}{\Longmapsto} \Delta_w \quad , \forall w \in ID_{\mathcal{W}} \quad .$$

A special waveform with the designated name $w_\alpha \in ID^W$ is allowed to be "stubbed", i.e.

$$\alpha\Delta_{w_\alpha} \equiv_{Def} \overset{\phi^1_{w_\alpha}}{\longmapsto}$$

$$respectively \quad \alpha\Delta_{w_\alpha} \equiv_{Def} \overset{\phi^1_{w_\alpha}}{\Longmapsto} \langle \phi^2_{w_\alpha} \rangle \quad .$$

where $\phi^2_{w_\alpha}$ is another Boolean formula. This waveform is needed in order to define the activation condition in the same general way as for LSTD (cf. def. 6.3).

A bundle $\mathcal{W}$ of waveforms induces a set

$$\mathit{SEVENTS}_{\mathcal{W}} \equiv_{Def} \{\langle w, i\rangle \;\mid w \in \mathit{ID}^{W}, 1 \leq i \leq |\alpha\Delta_w|\}$$

of so–called symbolic events of $\mathcal{W}$ (abbreviated sevents *or simply events). The special event $\langle w_\alpha, 0\rangle$ is called the activation event of $\mathcal{W}$.*

$|\alpha\Delta_w|$ is the length of the LSTD–diagram $\alpha\Delta_w$. If the body of $\alpha\Delta_w$ is Δ_w,

$$\Delta_w \equiv_{Def} \Delta E_1 \Delta E_2 \ldots \Delta E_{k_w} \quad (|\alpha\Delta_w| = k_w)$$

then $\langle w, i\rangle$ refers to the phase ΔE_i for $i>0$; the special notation $\langle w, 0\rangle$ refers to the activation specification of $\alpha\Delta_w$.

The phases of Δ_w are required to have no additional liveness requirement*, i.e. each phase ΔE_i has the form*

$$\Delta E_i \equiv_{Def} \underline{\;\phi_1^i \;\overset{\phi_2^i}{\mid}\; \langle\phi_3^i\rangle\;} \quad .$$

Two further requirements are made on the static semantics of waveform events: First*, all phases are required to be* deterministic *and* complete*, i.e.*

$$\phi_2^i \Rightarrow \neg\phi_1^i, \quad \forall j = 1 \ldots k_w \qquad (1)$$

$$\phi_1^i \vee \phi_2^i \vee \phi_3^i \sim \mathbf{true}, \quad \forall j = 1 \ldots k_w \quad . \qquad (2)$$

Second*, each stable condition of a phase must be identical to the enabling condition of the preceding phase, i.e.*

$$\phi_1^i \sim \phi_2^{i-1}, \forall i = 1 \ldots k_w \quad ;$$

in particular, the stable–condition ϕ_1^1 must be equivalent to the activation specification of the waveform Δ_w, i.e.

$$\phi_1^1 \sim \phi_w^1 \equiv_{Def} \phi_2^0 \quad .$$

An STD–diagram $W\Delta$ over V is a bundle $\mathcal{W}$ of waveforms over V equipped with additional functional elements. It is defined as a structure

$$W\Delta \equiv_{Def} (\mathcal{W}, \mathit{XPOS}, \mathit{Prec}^{!}, \mathit{Conf}^{!}, \mathit{Prec}^{?}, \mathit{Conf}^{?}, \mathit{Leadsto})$$

where

$$XPOS_{W\Delta} \ : \ (e \in SEVENTS_{\mathcal{W}} \mapsto n_e \in N_0) \quad ;$$

n_e is called the X–position of sevent e. The activation event of each waveform has the X–position 0. Inner sevents have a positive X–position, which is strictly monotonic increasing along the index position $\langle w, i\rangle$ for each waveform w, $i = 1 \ldots |\alpha\Delta_w|$. $Prec^{\mu}_{W\Delta}$ and $Conf^{\mu}_{W\Delta}$ are binary relations on the set $SEVENTS_{\mathcal{W}}$, where μ is a mode tag, which is either ! (for a requirement), or ? (for an expectation). $Conf^{\mu}_{W\Delta}$ is a symmetric relation (for $\mu \in \{?, !\}$), i.e.

$$(e_1, e_2) \in Conf^{\mu}_{W\Delta} \Rightarrow (e_2, e_1) \in Conf^{\mu}_{W\Delta} \quad .$$

The elements of $Conf^{\mu}_{W\Delta}$ are written in set notation, i.e. $\{e_1, e_2\} \in Conf^{\mu}_{W\Delta}$. $Leadsto_{W\Delta}$ is a binary relation on the product $SEVENTS^0_{\mathcal{W}} \times SEVENTS_{\mathcal{W}}$, where

$$SEVENTS^0_{\mathcal{W}} =_{Def} SEVENTS_{\mathcal{W}} \cup \{\langle w_\alpha, 0\rangle\}.$$

The constraints represented by elements of the three asymmetric relations $Prec^{!}_{W\Delta}, Prec^{?}_{W\Delta}$ and $Leadsto_{W\Delta}$ are required not to go "backwards", i.e. for a relation $R \in \{Prec^{!}_{W\Delta}, Prec^{?}_{W\Delta}, Leadsto_{W\Delta}\}$

$$(e_1, e_2) \in R \Rightarrow n_{e_2} \geq n_{e_1} \quad .$$

The idea of having designated sorts of ?–constraints is similar to the concept of exit–conditions in the LSTD formalism. These constraints (which are also called *weak* constraints) allow to express an *expected* order of events on those signals which are controlled by the environment of a component interface.

The elements of the sets $Prec^{!}, Conf^{!}, Prec^{?}, Conf^{?}, Leadsto$ represent the classes of strong precedence, strong conflict, weak precedence, weak conflict, and leadsto–constraints, respectively.

STD–diagrams can be conveniently denoted in a tabular notation of the following form:

⟨STD-diagram-name⟩	
[optional:]	$\alpha\Delta_{w_\alpha}$
⟨wf-name-1⟩	$\alpha\Delta_1$
...	...
⟨wf-name-k⟩	$\alpha\Delta_k$
$Prec^!$	$\{\ldots\}$
$Conf^!$	$\{\ldots\}$
$Prec^?$	$\{\ldots\}$
$Conf^?$	$\{\ldots\}$
$Leadsto$	$\{\ldots\}$

In STD, the X–position has no relevance for the semantics; therefore, the definition of *XPOS* will in the following be omitted. Note however, that an extension of STD exists (called STDx) which associates the X–position with a clock–cycle count of a synchronous design.

Example 6.4 (Tabular representation of STD–diagram) *The diagram shown in figure 6.3 is represented by the following tabular notation :*

TL_P_and_not_Q_unless_Q

		Induced sevents:
v	$\langle\!\lvert not(Q) \rvert\!\rangle$ $\langle\!\lvert Q \rvert\!\rangle$	$\{\langle v, 0\rangle, \langle v, 1\rangle\}$
w	$\langle\!\lvert P \rvert\!\rangle$ $\langle\!\lvert not(P) \rvert\!\rangle$	$\{\langle w, 0\rangle, \langle w, 1\rangle\}$

$Prec^!$	$\{(\langle v, 1\rangle, \langle w, 1\rangle)\}$
$Conf^!$	$\emptyset$
$Prec^?$	$\emptyset$
$Conf^?$	$\emptyset$
$Leadsto$	$\emptyset$

Empty constraint sets will be omitted in forthcoming examples.

For technical reasons, it is convenient to consider each waveform to be "closed" at the end by addition of a special pseudo–sevent

$$\Delta E^{\top} \equiv_{Def} \xrightarrow[\textbf{true}]{\textbf{false}} \dashv \quad .$$

The effect of this "top–event" is that it can never be matched. The top event has a special graphical denotation:

$$\Delta E^{\top} \equiv_{Def} \longrightarrow\!\dashv \quad .$$

From example 6.1 we know that a top event can be added to the end of a LSTD–diagram without changing the semantics (note that **true** $\sim \neg$**false**).

Example 6.5 (Tabular representation of STD–diagram with top–event completion) *The diagram shown in figure 6.3 is represented after waveform completion by the following tabular notation :*

TL_P_and_ not_Q_unless_Q		*Induced sevents:*
v	$\langle\!\lvert not(Q) \rvert\!\rangle$ $\langle\!\lvert Q \rvert\!\rangle$	$\{\langle v,0\rangle, \langle v,1\rangle, \langle v,2\rangle\}$
w	$\langle\!\lvert P \rvert\!\rangle$ $\langle\!\lvert not(P) \rvert\!\rangle$	$\{\langle w,0\rangle, \langle w,1\rangle, \langle w,2\rangle\}$
$Prec^{!}$	$\{(\langle v,1\rangle, \langle w,1\rangle)\}$	

In the following, we will assume that all waveforms of an STD–diagram are extended by top–events.

6.2.2 Activation mode of STD–diagrams

Recall from the definition of LSTD–diagrams, that there are two so–called activation modes: *Invariant* activation (*Whenever* ...), and *initial* activation (*Initially*, ...).

The same concept is used for STD–diagrams.

Definition 6.2 (Activation mode) *Assume an assertion language* $A\mathcal{L}$*, a*

set V of variables, and an STD–diagram $W\Delta$ over V with a bundle of waveforms $\mathcal{W} = (w \in ID_{\mathcal{W}} \mapsto \alpha\Delta_w)$.

We define a function actmode *of $W\Delta$ as follows:*

$actmode(W\Delta) \equiv_{Def}$ *invariant* **iff** *all waveforms of $W\Delta$ have the form:*

$$\alpha\Delta_w \equiv_{Def} \overset{\phi_w^1}{\vdash\!\!\!-\!\!\!-} \Delta_w$$

$actmode(W\Delta) \equiv_{Def}$ *initial* **iff** *all waveforms of $W\Delta$ have the form:*

$$\alpha\Delta_w \equiv_{Def} \overset{\phi_w^1}{\vdash\!\!\!\!=} \Delta_w$$

Note that by definition 6.1 there are no further possibilities for the waveforms in a STD–diagram, hence the function actmode *is well–defined for all diagrams $W\Delta$.*

The concept of an activation specification is also derived from the corresponding concept in LSTD:

For both types of STD–diagrams (invariant or initial), the activation specification is the conjunction of the activation specifications of the waveforms of the diagram.

Note that in the case of initial–type diagrams, by definition 6.1 the only waveform allowed to have an initial–exit specification is the special waveform w_α.

Definition 6.3 (Activation specification) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and an STD–diagram $W\Delta$ over V with a bundle of waveforms $\mathcal{W} = (w \in ID_{\mathcal{W}} \mapsto \alpha\Delta_w)$.*

We define a function actspec *of $W\Delta$ as follows:*

- *case* $actmode(W\Delta) \equiv_{Def}$ *invariant :*

  Assume that

$$\alpha\Delta_w \equiv_{Def} \overset{\phi_w^1}{\vdash\!\!\!-\!\!\!-} \Delta_w \quad , \forall w \in ID_{\mathcal{W}} \quad .$$

Then

$$actspec(W\Delta) \equiv_{Def} \bigwedge_{w \in ID_W} \phi_w^1 \quad .$$

- *case* $actmode(W\Delta) \equiv_{Def}$ *initial :*

  Assume that

$$\alpha\Delta_w \equiv_{Def} \overset{\phi_w^1}{\Vdash\!\!-} \Delta_w \quad ,\forall w \in ID_{\mathcal{W}} \quad .$$

Then

$$actspec(W\Delta) \equiv_{Def} \bigwedge_{w \in ID_W} \phi_w^1$$

For initial*–type diagrams, we also define a function* $actexitspec(W\Delta)$*, as follows: If a special waveform exists of the form*

$$\alpha\Delta_{w_\alpha} \equiv_{Def} \overset{\phi^1}{\Vdash\!\!-} \langle\phi^2\rangle$$

then

$$actexitspec(W\Delta) \equiv_{Def} \phi^2$$

otherwise,

$$actexitspec(W\Delta) \equiv_{Def} \textbf{false} \quad .$$

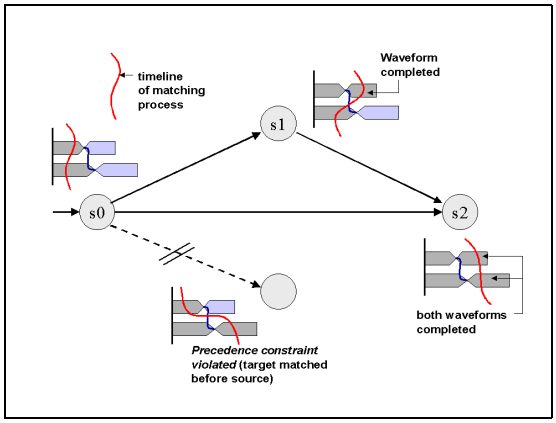
6.3 Semantics of STD–diagrams

The semantics of a STD–diagram $W\Delta$ is defined similarly to the semantics of an LSTD–diagram $\alpha\Delta$ in two steps:

1. From the body of the waveforms of the diagram, a POSA is derived by evaluation of the parallel matching progress on waveforms, considering the effect imposed by constraints during the construction;

2. the final diagram semantics is a LTL–formula, which combines the characterizing formula of the POSA constructed in step 1 with the activation semantics (either *initial* or *invariant*) of the diagram.

Figure 6.4: Illustration of the matching process for diagram *TL_P_and_not_Q_unless_Q*.



6.3.1 Derivation of SA from STD–body

As illustration of the idea, consider figure 6.4, which appeared earlier in chapter 2.

The figure 6.4 shows how the "timeline" associated with the matching process moves over the diagram from the left side (the activation bar) to the end of the waveforms. A waveform is said to be *completed* (with respect to the matching process), if the last sevent of the waveform has been matched, i.e. if the timeline has reached the top–event of that waveform.

The position of a timeline can be characterized by the set of events which have already been matched (the *past–set*), and the set of events which have not yet been matched (the *future–set*).

The following table defines the position of the timeline for each of the nodes displayed in figure 6.4.

node	*past–set*	*future–set*
s0	$\{\langle v,0\rangle, \langle w,0\rangle\}$	$\{\langle v,1\rangle, \langle v,2\rangle, \langle w,1\rangle, \langle w,2\rangle\}$
s1	$\{\langle v,0\rangle, \langle v,1\rangle, \langle w,0\rangle\}$	$\{\langle v,2\rangle, \langle w,1\rangle, \langle w,2\rangle\}$
s2	$\{\langle v,0\rangle, \langle v,1\rangle, \langle w,0\rangle, \langle w,1\rangle\}$	$\{\langle v,2\rangle, \langle w,2\rangle\}$

The next definition associates a SA to the body of a STD–diagram.

Definition 6.4 (SA of STD–body) *Assume an assertion language $A\mathcal{L}$, a set V of variables , and an STD–diagram $W\Delta$ over V,*

$$W\Delta \equiv_{Def} (\mathcal{W}, Prec^{!}, Conf^{!}, Prec^{?}, Conf^{?}, Leadsto) \quad .$$

Define the body of $W\Delta$ to be the set

$$W\Delta' \equiv_{Def} \{\Delta_v \mid \Delta_v \text{ is the body of } \alpha\Delta_v \ , \alpha\Delta_v = \mathcal{W}(v) \text{ for some } \ v \in ID_{\mathcal{W}}\} \quad .$$

We define the components of an SA $\mathcal{A}$ over V, $\mathcal{A} \equiv_{Def} \mathcal{A}_{W\Delta'}$ in the rest of this section.

Definition of $\mathcal{A}_{W\Delta'}$. The SA $\mathcal{A}_{W\Delta'}$ is defined by

$$\mathcal{A}_{W\Delta'} \equiv_{Def} (V, Locs, Edges, \{\zeta_0\}, F_{W\Delta})$$

where the set of locations *Locs* contains the set of timeline positions encountered during the matching process of diagram $W\Delta$. The set *Edges* is a union of three sets,

$$Edges \equiv_{Def} Edges^{u} \cup Edges^{e} \cup Edges^{s} \quad .$$

The definition is given in three steps:

1. definition of the set *Locs* of *reachable timelines*,
2. definition of the sets $Edges^{u}$, $Edges^{e}$ and $Edges^{s}$ of "unwinding"–, "exit"– and "stuttering"–transitions, and
3. definition of the set $F_{W\Delta}$ of acceptance states.

Definition of the set *Locs* . Each "location" of the SA $\mathcal{A}_{W\Delta}$ describes the current status of a matching process using the concept of a timeline position.

Definition 6.5 (Timeline position) *A timeline position, denoted by ζ, is a mapping*

$$\zeta \ : \ (v \in ID_{\mathcal{W}} \mapsto \langle v, i\rangle \in SEVENTS^{\top}_{W\Delta}) \quad .$$

where

$$SEVENTS^{\top}_{W\Delta} \equiv_{Def} SEVENTS_{W\Delta} \cup \{\langle v, |\alpha\Delta_v| + 1\rangle \mid v \in ID_{\mathcal{W}}\}$$

is the set of inner sevents plus the set of added top–events of the diagram $W\Delta$*. Note that after extension, each waveform* v *has at least one sevent (*$\langle v, 1\rangle$*).*

A *transition* from a timeline ζ to another timeline ζ' happens, if one or more sevents in the set $E^{\star}_{\zeta}$ of events *enabled at* ζ are matched; this set is defined by

$$E^{\star}_{\zeta} \equiv_{Def} \{\zeta(v) \mid v \in ID_{\mathcal{W}}, \zeta(v) \neq \zeta^{\top}(v)\} \quad ;$$

note that top–events are excluded from the set $E^{\star}_{\zeta}$. The set $\wp E^{\star}_{\zeta}$ of *non–empty subsets of events which are enabled at* ζ is defined by

$$\wp E^{\star}_{\zeta} \equiv_{Def} \wp(E^{\star}_{\zeta}) \backslash \emptyset \quad .$$

Elements of $\wp E^{\star}_{\zeta}$ are typically denoted by letter $\mathcal{E}$.

The progress to a *successor timeline* ζ' induced by matching an event set $\mathcal{E} \in E^{\star}_{\zeta}$ at position ζ is denoted by $\zeta + \mathcal{E}$, which is defined by

$$\zeta + \mathcal{E} \quad \equiv_{Def} \quad (v \mapsto \begin{array}{ll} \zeta(v) & \text{if } \zeta(v) \notin \mathcal{E} \quad , \\ \langle v, i+1\rangle & \text{if } \zeta(v) \in \mathcal{E} \text{ and } \zeta(v) = \langle v, i\rangle \quad) \end{array}$$

The next definitions are fundamental to the idea of the matching process.

Definition 6.6 (Successor timeline) *Let* ζ, ζ' *be timelines, i.e. mappings from* $ID_{\mathcal{W}}$ *to* $SEVENTS^{\top}_{W\Delta}$*.*

Then ζ' *is defined to be a successor of* ζ*, denoted by* $\zeta \rightarrow \zeta'$*, as follows:*

$$\zeta \rightarrow \zeta' \textbf{ iff } \exists \mathcal{E} \in \wp E^{\star}_{\zeta} \,.\, \zeta' = \zeta + \mathcal{E} \quad .$$

A transition from ζ *to a successor* ζ' *due to a match of the events in* $\mathcal{E}$ *is also denoted more specifically by*

$$\zeta \xrightarrow{\mathcal{E}} \zeta' \quad .$$

Note that for a timeline position ζ, $\zeta + \mathcal{E}$ is again a timeline position for $\mathcal{E} \in \wp E^{\star}_{\zeta}$; in particular,

$$(\zeta + \mathcal{E})(v) \in SEVENTS^{\top}_{W\Delta}, \forall v \in ID_{\mathcal{W}} \quad .$$

This is the case, because top–events are by definition excluded from the set $E^{\star}_{\zeta}$.

Definition 6.7 (Partial order of timelines) *Define a partial order $\preceq_{\mathcal{W}}$ on the set $SEVENTS^{\top}_{\mathcal{W}}$ by:*

$$\langle v, k\rangle \preceq_{\mathcal{W}} \langle v', k'\rangle \textbf{ iff } v = v' \wedge k \leq k' \quad .$$

The partial order $\preceq_{\mathcal{W}}$ on the set of sevents induces a partial order on the set of timelines, denoted by $\preceq$:

$$\zeta \preceq \zeta' \textbf{ iff } \forall v \in ID_{\mathcal{W}} \ : \ \zeta(v) \preceq_{\mathcal{W}} \zeta'(v) \quad .$$

It is easy to see that $\preceq_{\mathcal{W}}$ and $\preceq$ are indeed partial orders (i.e. reflexive and transitive).

The partial order $\preceq$ has a unique minimal element ζ_0 and a unique maximal element $\zeta^{\top}$(called the *top–timeline*), defined by

$$\zeta_0 \equiv_{Def} (v \mapsto \langle v, 1\rangle) \quad , v \in ID_{\mathcal{W}}$$

respectively

$$\zeta^{\top} \equiv_{Def} (v \mapsto \langle v, |\alpha\Delta_v| + 1\rangle) \quad , v \in ID_{\mathcal{W}} \quad .$$

A timeline ζ *cuts* a precedence–constraint, if the target of the constraint has been matched, but the source has not been matched yet.

A timeline ζ *cuts* a leadsto–constraint, if the source of the constraint has been matched, but the target has not been matched yet.

Definition 6.8 (Constraint cut of a timeline) *Given a timeline ζ, define*

$$\begin{array}{lll} past(\zeta) & \equiv_{Def} & \{\langle v, i\rangle \mid v \in ID_{\mathcal{W}}, 1 \leq i < j_v \text{ if } \zeta(v) = \langle v, j_v\rangle\} \text{ and} \\ future(\zeta) & \equiv_{Def} & SEVENTS^{\top}_{W\Delta} \backslash past(\zeta) \quad . \end{array}$$

Let $\rightarrow_c \in Prec^{!} \cup Prec^{?}, \rightarrow_c \equiv_{Def} (e_1, e_2)$.

The fact that ζ cuts a precedence–constraint, denoted by $\zeta \not\mapsto_c$, is defined by:

$$\zeta \not\mapsto_c \quad \textbf{iff } e_2 \in past(\zeta) \wedge e_1 \in future(\zeta) \quad .$$

Let $\rightsquigarrow_c \in Leadsto, \rightsquigarrow_c \equiv_{Def} (e, e')$.
The fact that ζ cuts a leadsto–constraint, denoted by $\zeta \not\rightsquigarrow_c$, is defined by:

$$\zeta \not\rightsquigarrow_c \quad \textbf{iff } e \in past(\zeta) \wedge e' \in future(\zeta) \quad .$$

While the fact that a precedence– or leadsto–constraint is cut by a timeline ζ can be attributed to the timeline reached after matching a set $\mathcal{E}$ of sevents, a violation of a conflict constraint is attributed to the set $\mathcal{E}$.

Definition 6.9 (Conflict–free set of sevents) *Let ζ be a timeline, $\mathcal{E} \in \wp E^\star_\zeta$.*

The fact that a set $\mathcal{E}$ is called conflict–free, denoted $\mathcal{E}^\surd$, is defined by

$$\mathcal{E}^\surd \textbf{ iff } R^\#(\mathcal{E}) \cap (Conf^! \cup Conf^?) = \emptyset$$

where the definition of the (irreflexive and symmetric) conflict–relation $R^\#(\mathcal{E})$ induced by set $\mathcal{E}$ is

$$R^\#(\mathcal{E}) \equiv_{Def} \{(e_1, e_2) \mid e_1 \neq e_2 \text{ and } \{e_1, e_2\} \subseteq \mathcal{E}\} \ , \mathcal{E} \subseteq SEVENTS^\top_{W\Delta} \quad .$$

Definition 6.10 (Unwinding successor) *Let ζ' be a successor of ζ. ζ' is called an unwinding–successor of ζ, denoted by $\zeta \stackrel{\surd}{\Longrightarrow} \zeta'$,* **iff**

1: $\neg(\exists \rightarrow_c \in Prec^! \cup Prec^? \ : \ \zeta' \not\mapsto_c)$
2: $\exists \mathcal{E} \in \wp E^\star_\zeta \ . \ \mathcal{E}^\surd \wedge \zeta \stackrel{\mathcal{E}}{\longrightarrow} \zeta' \quad .$

Condition 1 means, that ζ' does not cut any precedence constraint, and condition 2 means that step $\mathcal{E}$ is a conflict–free match, which leads from ζ to ζ'.

Given these definitions, we can next define the set of timelines *reachable from the initial timeline* ζ_0 as follows:

$$Locs =_{Def} \{\zeta \mid \zeta_0 \stackrel{\surd}{\Longrightarrow}{}^{*} \zeta\} \quad .$$

We define the set of "unwinding" transitions, denoted $Trans^u$, by

$$Trans^u \equiv_{Def} \{\zeta \stackrel{\mathcal{E}}{\longrightarrow} \zeta' \mid \zeta, \zeta' \in Locs, \zeta' = \zeta + \mathcal{E}, \mathcal{E} \in \wp E^{\star}_{\zeta}, \mathcal{E}^{\surd}\} \quad .$$

An unwinding transition is caused by a conflict–free match, which leads from a reachable timeline to an (again reachable) successor timeline.

Definition of the set $Edges^u$. The set $Edges^u$ is a functional image of the set of unwinding transition $Trans^u$, referring to the contents of the phases of the waveforms of the diagram $W\Delta$.

Recall the structure of an STD–diagram described in definition 6.1.

If the body of a waveform named v is Δ_v,

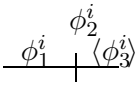
$$\Delta_v \equiv_{Def} \Delta E_1 \Delta E_2 \ldots \Delta E_{k_v}$$

then the waveform extended with top–event is defined by

$$\Delta_v \equiv_{Def} \Delta E_1 \Delta E_2 \ldots \Delta E_{k_v} \Delta E_{k_v+1} \quad ;$$

$\langle v, i\rangle$ refers to the phase ΔE_i , for $1 \leq i \leq k_v + 1$. If

$$\Delta E_i \equiv_{Def}$$



then we define

$$\begin{array}{lll} stab(\langle v, i\rangle) & \equiv_{Def} \quad \phi_1^i & \text{(stable condition)}, \\ enab(\langle v, i\rangle) & \equiv_{Def} \quad \phi_2^i & \text{(enable condition)}, \\ exit(\langle v, i\rangle) & \equiv_{Def} \quad \phi_3^i & \text{(exit condition)} \quad . \end{array}$$

An unwinding transition corresponding to a match of some event set $\mathcal{E}$ may occur, if all enable conditions of the sevents of set $\mathcal{E}$ are satisfied. For those sevents of a timeline, which are not matched during an unwinding transition, the stable condition must be satisfied.

This leads to the following definition of the set $Edges^u$:

$$Edges^u \equiv_{Def} \{(\zeta, \phi^{\mathcal{E}}_{\zeta}, \zeta') \mid \exists \mathcal{E} \in \wp E^{\star}_{\zeta} \,.\, (\zeta \xrightarrow{\mathcal{E}} \zeta') \in Trans^u\}$$

where $\phi^{\mathcal{E}}_{\zeta} \equiv_{Def}$ is defined in the next definition.

Definition 6.11 (Unwinding condition) *The condition $\phi^{\mathcal{E}}_{\zeta}$ is defined as*

$$\phi^{\mathcal{E}}_{\zeta} \equiv_{Def} \bigwedge_{v \in ID_{\mathcal{W}}, \zeta(v) \notin \mathcal{E}} stab(\zeta(v)) \quad \wedge \quad \bigwedge_{v \in ID_{\mathcal{W}}, \zeta(v) \in \mathcal{E}} enab(\zeta(v)) \quad .$$

Definition of the set $Edges^e$. Recall from the definition of the semantics of LSTD–diagrams, that two sorts of transitions exist: First, the set of unwinding transitions, and second the set of transitions due to an "exit" condition of an sevent.

In the case of LSTD–diagrams, an sevent–exit happens, if the exit–condition of a LSTD–phase is satisfied and neither the stable– nor the enable–condition of the phase is satisfied.

The situation is more complicated with STD–diagrams: An exit may also occur due to violation of a weak constraint.

Therefore, $Edges^e$ is a union two sets:

$$Edges^e \equiv_{Def} Edges^{we} \cup Edges^{ce}$$

where $Edges^{we}$ contains the set of so–called waveform–exit–edges, and $Edges^{ce}$ contains the set of so–called (weak–)constraint–exit–edges.

We first consider the definition of set $Edges^{we}$. At a current timeline position ζ, a *waveform exit* is defined to occur, if on at least one waveform an sevent–exit occurs. An exit causes a direct move from the current timeline position to the top–timeline $\zeta^{\top}$.

This leads to the following definition of the set $Edges^{we}$:

$$Edges^{we} \equiv_{Def} \{(\zeta, \phi^{we}_{\zeta}, \zeta^{\top}) \mid \zeta \in Locs \backslash \{\zeta^{\top}\}} \quad ,$$

where

$$\phi^{we}_{\zeta} \equiv_{Def} \bigvee_{v \in ID_{\mathcal{W}}} (\; exit(\zeta(v)) \wedge \neg stab(\zeta(v)) \wedge \neg enab(\zeta(v)) \;) \quad .$$

Weak constraint violation. Weak constraint violation happens, if either a weak conflict or a weak precedence constraint is violated.

As preparation, we need the definition of weak–conflict and weak–precedence violation.

Definition 6.12 (Weak–conflict set of sevents) *Let ζ be a timeline, $\mathcal{E} \in \wp E^\star_\zeta$.*

The fact that a set $\mathcal{E}$ has a weak–conflict, denoted $\mathcal{E}^{w\#}$, is defined by

$$\mathcal{E}^{w\#} \textbf{ iff } R^\#(\mathcal{E}) \cap Conf^? \neq \emptyset \quad .$$

Definition 6.13 (Constraint–exit successor) *Let ζ' be a successor of ζ. ζ' is called a constraint–exit–successor of ζ, denoted by $\zeta \stackrel{w\#}{\Longrightarrow} \zeta'$, **iff** (at least) one of the following conditions holds:*

$$\begin{array}{ll} a: & (\exists \rightarrow_c \in Prec^? \ : \ \zeta' \not\rightarrow_c) \\ b: & \exists \mathcal{E} \in \wp E^\star_\zeta \ . \ \mathcal{E}^{w\#} \wedge \zeta \stackrel{\mathcal{E}}{\longrightarrow} \zeta' \end{array} \quad .$$

Condition 1 means, that ζ' cuts some weak precedence constraint, and condition 2 means that step $\mathcal{E}$ is a match which has a weak–conflict, and leads from ζ to ζ'.

The concept of a constraint–exit–successor is mainly used for technical simplicity. In fact, a step to an exit successor means termination of the matching process, with positive (accepting) result.

This semantics can be described easily in the way that all exit–successor timelines *are identified with the top–timeline* $\zeta^\top$.

This motivates the following definition of the set $Trans^{ce}$ of transitions corresponding to violation of a weak constraint:

$$Trans^{ce} \equiv_{Def} \{\zeta \stackrel{\mathcal{E}}{\longrightarrow} \zeta^\top \mid \zeta \in Locs, \exists \mathcal{E} \in E^\star_\zeta \ . \ \zeta \stackrel{w\#}{\Longrightarrow} \zeta + \mathcal{E}\} \quad .$$

The set $Edges^{ce}$ of edges is defined as a functional image of the set of transitions $Trans^{ce}$; the definition of the set $Edges^{ce}$ is as follows:

$$Edges^{ce} \equiv_{Def} \{(\zeta, \phi^{\mathcal{E}}_\zeta, \zeta^\top) \mid \exists \mathcal{E} \in E^\star_\zeta \ . \ (\zeta \stackrel{\mathcal{E}}{\longrightarrow} \zeta^\top) \in Trans^{ce}\}$$

where $\phi^{\mathcal{E}}_\zeta$ is defined as for $Edges^u$ (see definition 6.11).

Construction of the set $Edges^s$. The set $Edges^s$ is a functional image of the set of reachable timeline locations, defined by

$$Edges^s \equiv_{Def} \{(\zeta, \phi^s_\zeta, \zeta) \mid \zeta \in Locs\}$$

where ϕ^s_ζ is defined in the next definition.

Definition 6.14 (Stable condition) *The condition ϕ^s_ζ is defined as*

$$\phi^s_\zeta \equiv_{Def} \bigwedge_{v \in ID_{\mathcal{W}}} stab(\zeta(v)) \quad .$$

Construction of the set $F_{W\Delta}$. Intuitively, a timeline describes either a "stable" or an "instable" situation. A situation ζ is called *stable*, if an infinite repetition of states, which satisfy the stable condition ϕ^s_ζ, is an accepted behavior.

The distinction is made by the class of so–called leadsto–constraints. If a timeline "cuts" a leadsto–constraint, than the situation is called *instable*. The term "instable" means here that something must happen which leads out of the instable situation. The only possibility to leave an instable situation is to have a progress of the timeline (corresponding to a matching of sevents).

This intuition is captured by the following definition of the set $F_{W\Delta}$ of *stable situations*:

$$F_{W\Delta} \equiv_{Def} Locs \backslash \{\zeta \in Locs \mid \exists \langle v, k\rangle \in past(\zeta), \langle v', k'\rangle \in future(\zeta) \; . \; (\langle v, k\rangle, \langle v', k'\rangle) \in Leadsto_{W\Delta}\}$$

Note that the top–timeline $\zeta^\top$ *always* describes a stable situation, because it consists of the added top–events (which can not cut any constraint).

6.3.2 Definition of semantics of STD–diagram

The definition of the semantics of STD–diagrams is analogous to the definition of the semantics of LSTD diagrams, namely a combination of a declarative definition and the semantics defined for Symbolic Automata.

Definition 6.15 (STD–diagram semantics) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and an STD–diagram $W\Delta$ over V. Then the semantics of $W\Delta$, denoted by $L(W\Delta)$, is defined as follows:*

- *case* $actmode(W\Delta) \equiv_{Def}$ *invariant : In this case,*

$$L(W\Delta) =_{Def} \{\sigma \in Comp(V) \mid \quad \forall k \geq 0 : \\ \sigma(k) \models actspec(W\Delta) \rightarrow \sigma^{(k+1)} \in L(\Delta)\}$$

- *case* $actmode(W\Delta) \equiv_{Def}$ *initial : In this case,*

$$L(W\Delta) =_{Def} \{\sigma \in Comp(V) \mid \quad \sigma(0) \models actexitspec(W\Delta) \text{ or} \\ \sigma(0) \models actspec(W\Delta) \wedge \sigma^{(1)} \in L(\Delta)\}$$

where $L(\Delta) \equiv_{Def} L(\mathcal{A}_{W\Delta})$ *is the semantics of the SA* $\mathcal{A}$ *obtained from* $W\Delta$ *according to definition 6.4.*

6.4 Translation of STD–diagrams to temporal logic

In the beginning of this chapter, we motivated the concept of STD as a kind of parallel composition of LSTD–diagrams.

The definitions of STD and LSTD are "incompatible", because the definition of STD does not allow "non–deterministic" sevents (cf. definition 6.1). On the other hand, e.g. the possibility to express preemptive behaviour using weak constraints appears to increase the expressive power of STD over LSTD.

Therefore, the expressive power of STD and LSTD seems to be unrelated. Interestingly, this is not the case, due to the following line of reasoning:

1. The semantics of a STD–body can be characterized by a stuttering–invariant and deterministic POSA.

2. Due to theorem 5.1, the semantics of a deterministic POSA can be characterized by an equivalent set ΔS of LSTD bodies.

3. The activation semantics is defined in the same way for STD and LSTD; hence, the semantics of a STD–diagram can be characterized by an equivalent LSTD–specification *SPEC*.

In this section we will establish the first step, i.e. we investigate the structure of the SA derived from an STD–body in detail.

6.4.1 Properties of the characterization of STD–body by SA

The next lemma is the key to the characterization of the semantics of STD–diagrams in terms of temporal logic.

Lemma 6.1 (POSA of STD–body) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and an STD–diagram $W\Delta$ over V, with body $W\Delta'$. Then the SA $\mathcal{A}_{W\Delta'}$ constructed in 6.4 is a POSA.*

Proof of lemma 6.1 . Recall that according to definition 3.7, a partially ordered symbolic automaton (POSA) $\mathcal{A}$ over V is a SA

$$\mathcal{A} \; : \; (V, \mathit{Locs}, \mathit{Edges}^{po}, L_0, F)$$

with the following restriction on the set Edges^{po}: Define a binary relation $\rightarrow$ on the set *Locs* (denoted in infix notation) by

$$\ell_1 \rightarrow \ell_2 \ \textbf{iff} \ \exists \phi \ . \ (\ell_1, \phi, \ell_2) \in Edges^{po} \quad .$$

Then the SA $\mathcal{A}$ is called a partially ordered SA, iff the relation $\rightarrow^*$ (the reflexive, transitive closure of $\rightarrow$) is a partial order. In particular, it is required to be anti–symmetric, i.e.:

$$(\ell_1 \rightarrow^* \ell_2) \wedge (\ell_2 \rightarrow^* \ell_1) \quad \Longrightarrow \quad \ell_1 = \ell_2 \quad .$$

According to definition 6.4, the SA $\mathcal{A}_{W\Delta'}$ is defined by

$$\mathcal{A}_{W\Delta'} \equiv_{Def} (V, Locs, Edges, \{\zeta_0\}, F)$$

where the set *Edges* consists of the following edge types:

- unwinding edges: An unwinding edge corresponds to a transition $\zeta \xrightarrow{\mathcal{E}} \zeta'$ where matching the set $\mathcal{E}$ of events advances the timeline from ζ to ζ'; this implies by definition 6.7 that $\zeta \preceq \zeta'$.
- exit edges: an exit edge has always the form $(\zeta \xrightarrow{\mathcal{E}} \zeta^{\top})$; in this case, by definition $\zeta \preceq \zeta^{\top}$.
- stuttering (loop) edges: a stuttering edge has the form $(\zeta \longrightarrow \zeta)$; again, by definition, $\zeta \preceq \zeta$.

It follows that the relation $\rightarrow$ is a subset of the partial order $\preceq$; therefore, $\rightarrow^*$ must also be a partial order.

q.e.d.

An immediate consequence of lemma 6.1 is the fact, that theorem 3.3 can be applied as follows:

1. The automaton $\mathcal{A} \equiv_{Def} \mathcal{A}_{W\Delta'}$ derived from the STD–body $W\Delta'$ is a POSA;
2. by theorem 3.3 , there exists a formula $\phi_{\mathcal{A}} \in LTL_{A\mathcal{L}}$ over V such that

$$L(\mathcal{A}) = L(\phi_{\mathcal{A}}) \quad .$$

Our next goal is to show that the POSA derived from an STD–body is 1) deterministic and 2) stuttering invariant.

Theorem 6.1 (Properties of POSA derived from STD body) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and an STD–diagram $W\Delta$ over V, with body $W\Delta'$. Let $\mathcal{A} \equiv_{Def} \mathcal{A}_{W\Delta'}$ be the POSA derived from the STD–body $W\Delta'$.*

Then the following facts hold:

1. *$\mathcal{A}$ is stuttering–invariant, and*

2. *$\mathcal{A}$ is deterministic and complete.*

Proof of theorem 6.1 . Let $\mathcal{A} \equiv_{Def} (V, Locs, Edges, L_0, F)$. Recall the following abbreviation from theorem 3.3 : For each pair of locations $\ell, \ell' \in Locs$, $\ell \to \ell'$, let

$$\phi_{\ell,\ell'} =_{Def} \textbf{any } \phi \,.\, (\ell, \phi, \ell') \in Edges \quad .$$

Note that $\mathcal{A}$ is not normalized by construction.

The first criterion of normalization (the requirement that for all locations ℓ, $\ell \to \ell$, i.e. self–loops exist), is satisfied by construction.

The second requirement is that no parallel edges exist; this is *not* satisfied by construction, because there may be two parallel edges from any location ℓ to the top–location $\ell^\top$, corresponding to waveform–exit and to constraint–exit, respectively.

However, it can be assumed that these parallel edges are combined by joining the labels of the parallel edges into a disjunction.

We can thus assume that $\mathcal{A}$ is normalized. Hence $|\{\phi \mid (\ell, \phi, \ell') \in Edges\}| = 1$, which means that $\phi_{\ell,\ell'}$ is uniquely defined for all $\ell, \ell' \in Locs$.

For the two claims of the theorems, we have to establish the following propositions:

(1)	$\forall \ell \neq \ell', \ell \to \ell' \;:\; \phi_{\ell,\ell'} \Rightarrow \phi_{\ell',\ell'}$	$\mathcal{A}$ stuttering–invariant
(2.1)	$\forall \ell \neq \ell', \ell \to \ell' \;:\; \phi_{\ell,\ell} \Rightarrow \neg\phi_{\ell,\ell'}$	$\mathcal{A}$ deterministic
(2.2)	$\forall \ell'_1 \neq \ell'_2, \ell'_1 \neq \ell \neq \ell'_2,$	
	$\ell \to \ell'_1, \ell \to \ell'_2 \;:\; \phi_{\ell,\ell'_1} \Rightarrow \neg\phi_{\ell,\ell'_2}$	(disjoint successors)
(3)	$\forall \ell \;:\; \phi_{\ell,\ell} \vee \bigvee_{\ell \neq \ell', \ell \to \ell'} \phi_{\ell,\ell'} \sim \textbf{true} \quad .$	$\mathcal{A}$ complete

Proof of proposition (1). It is useful to consider what these tautologies mean with respect to the matching process associated with an STD body (cf. figure 6.4).

Condition (1) states that whenever an unwinding step occurs (due to matching a non–empty set of events), the same state–condition which allows the matching step will allow to *remain* in the state reached after the matching step, if the state–condition does not change.

This follows immediately from the definition 6.1 of STD–diagrams, which requires that the stable–condition of a phase ΔE must be identical to the trigger condition of the preceding phase.

We consider a transition from ζ (corresponding to location ℓ) to ζ' (corresponding to location ℓ') due to a match of event–set $\mathcal{E}$, i.e., $\zeta' = \zeta + \mathcal{E}$. Recall that the unwinding condition $\phi^{\mathcal{E}}_{\zeta}$ is defined in definition 6.11 as

$$\phi^{\mathcal{E}}_{\zeta} \equiv_{Def} \bigwedge_{v \in ID_{\mathcal{W}}, \zeta(v) \notin \mathcal{E}} stab(\zeta(v)) \quad \wedge \quad \bigwedge_{v \in ID_{\mathcal{W}}, \zeta(v) \in \mathcal{E}} enab(\zeta(v)) \quad ;$$

while the condition $\phi^{s}_{\zeta'}$ (the stable condition) is defined in definition 6.14 as

$$\phi^{s}_{\zeta'} \equiv_{Def} \bigwedge_{v \in ID_{\mathcal{W}}} stab(\zeta'(v)) \quad .$$

For $v \in ID_{\mathcal{W}}, \zeta(v) \notin \mathcal{E}$, $stab(\zeta'(v)) = stab(\zeta(v))$.

For $v \in ID_{\mathcal{W}}, \zeta(v) \in \mathcal{E}$, either $stab(\zeta'(v)) = enab(\zeta(v))$ or $stab(\zeta'(v)) =$ **true** , which is the case when the last event of a waveform has been matched (cf. example 6.5).

Thus, in any case it follows that

$$\phi_{\ell,\ell'} = \phi^{\mathcal{E}}_{\zeta} \Rightarrow \phi^{s}_{\zeta'} = \phi_{\ell',\ell'} \quad .$$

which proves that $\mathcal{A}$ is stuttering–invariant.

Proof of proposition (2.1). We have to show that the progress of the matching process is deterministic. With every change of the state–condition, exactly one of four possibilities exist:

1. either no matching happens (because the stable–condition is satisfied), or

2. a waveform exit condition is met, or

3. a match occurs which leads to a weak constraint exit , or

4. a match occurs which leads to an unwinding step.

We recall the relevant definitions:

- Stutter step:

A *stutter step* occurs in location ζ, if the actual state condition ρ satisfies the stable–condition ϕ^s_ζ , defined by

$$\phi^s_\zeta \equiv_{Def} \bigwedge_{v \in ID_{\mathcal{W}}} stab(\zeta(v)) \quad .$$

- Waveform exit:

A *waveform exit step* occurs in location ζ, if the actual state condition ρ satisfies ϕ^{we}_ζ , defined by

$$\phi^{we}_\zeta \equiv_{Def} \bigvee_{v \in ID_{\mathcal{W}}} (\ exit(\zeta(v)) \wedge \neg stab(\zeta(v)) \wedge \neg enab(\zeta(v)) \ ) \quad .$$

- Weak constraint violation exit or unwinding condition:

With respect to the matching process, weak constraint exit and unwind steps are similar; they are due to the match of an event set $\mathcal{E}$, satisfying the condition $\phi^{\mathcal{E}}_\zeta$, defined by

$$\phi^{\mathcal{E}}_\zeta \equiv_{Def} \bigwedge_{v \in ID_{\mathcal{W}}, \zeta(v) \notin \mathcal{E}} stab(\zeta(v)) \quad \wedge \quad \bigwedge_{v \in ID_{\mathcal{W}}, \zeta(v) \in \mathcal{E}} enab(\zeta(v)) \quad .$$

Proposition (2.1) states that a state–condition which allows to remain in a state reached after some matching step, does not allow a matching step at the same time.

First, confer ϕ^s_ζ against ϕ^{we}_ζ: $\rho \models \phi^{we}_\zeta$ implies that ρ does not satisfy on some waveform v the stable condition, i.e. $\rho \models \neg stab(\zeta(v))$. Thus, is is not possible that $\rho \models \phi^s_\zeta$.

Second, confer ϕ^s_ζ against $\phi^{\mathcal{E}}_\zeta$: $\rho \models \phi^{\mathcal{E}}_\zeta$ implies that ρ does satisfy on some waveform v the enable (trigger) condition, i.e. $\rho \models enab(\zeta(v))$. Due to the deterministic property of events of STD–waveforms, this implies that $\rho \models \neg stab(\zeta(v))$. Thus, it is not possible that $\rho \models \phi^s_\zeta$.

Thus, proposition (2.1) is proved.

Proof of proposition (2.2). It remains to prove proposition (2.2).

We have to demonstrate the disjointness of the "transition guards" $\phi_{\ell,\ell'}$, which correspond to either waveform exit or match condition (causing either an unwind step or weak constraint violation).

First, confer some matching condition $\phi^{\mathcal{E}}_\zeta$ against the waveform exit condition ϕ^{we}_ζ: $\rho \models \phi^{we}_\zeta$ implies that ρ does not satisfy on some waveform v the stable condition, i.e. $\rho \models \neg stab(\zeta(v))$ and $\rho \models \neg enab(\zeta(v))$. We have to consider two cases:

- case 1: $\zeta(v) \notin \mathcal{E}$. Then $\rho \not\models \phi^{\mathcal{E}}_\zeta$, because $\rho \not\models stab(\zeta(v))$.
- case 2: $\zeta(v) \in \mathcal{E}$. Then $\rho \not\models \phi^{\mathcal{E}}_\zeta$, because $\rho \not\models enab(\zeta(v))$.

Second, confer some matching condition $\phi^{\mathcal{E}_1}_\zeta$ against another matching condition $\phi^{\mathcal{E}_2}_\zeta$, $\mathcal{E}_1 \neq \mathcal{E}_2$. Consider some event $e \in \mathcal{E}_1 \backslash \mathcal{E}_2$, where $e \equiv_{Def} \zeta(v)$ for some waveform v.

Assume that $\rho \models \phi^{\mathcal{E}_1}_\zeta$, so by definition $\rho \models enab(\zeta(v))$; on the other hand, this implies $\rho \not\models \phi^{\mathcal{E}_2}_\zeta$, because $\rho \not\models stab(\zeta(v))$ follows from the definition of STD ($stab(\zeta(v)) \Rightarrow \neg enab(\zeta(v))$).

Thus, proposition (2.2) is proven.

Proof of proposition (3). We have to show that the matching process is complete, i.e. for each timeline ζ, state–condition ρ we have one of the following cases:

$$\rho \models \phi^s_\zeta \qquad (1)$$
$$\rho \models \phi^{we}_\zeta \qquad (2)$$
$$\exists \mathcal{E} \,.\, \rho \models \phi^{\mathcal{E}}_\zeta \qquad (3)$$

Assume that for some timeline ζ, state–condition ρ, condition (1) and (2) do not hold.

By (1), $\rho \models \neg\phi^s_\zeta$, so by definition of ϕ^s_ζ: $\exists v_0 \in ID_{\mathcal{W}} \,.\, \rho \models \neg stab(\zeta(v_0))$.

Define

$$\mathcal{E}_0 =_{Def} \{\zeta(v) \mid \rho \models \neg stab(\zeta(v)), v \in ID_{\mathcal{W}}\}$$

which is a non–empty set ($v_0 \in \mathcal{E}$).

We show that $\rho \models \phi_\zeta^{\mathcal{E}_0}$, i.e.

$$\forall v, v \in ID_{\mathcal{W}}, \zeta(v) \in \mathcal{E}_0 \ : \ \rho \models enab(\zeta(v)) \qquad \text{(i)}$$
$$\forall v, v \in ID_{\mathcal{W}}, \zeta(v) \notin \mathcal{E}_0 \ : \ \rho \models stab(\zeta(v)) \quad . \quad \text{(ii)}$$

The second fact (ii) follows from the definition of $\mathcal{E}_0$.

Consider (i): Assume that $\exists v_1, \zeta(v_1) \in \mathcal{E}_0 \ . \ \rho \models \neg enab(\zeta(v_1))$. Since (2) is assumed not to hold, $\rho \models \neg \phi_\zeta^{we}$, i.e.

$$\rho \models \neg(exit(\zeta(v_1)) \wedge \neg stab(\zeta(v_1)) \wedge \neg enab(\zeta(v_1))) \quad . \quad (*)$$

Since $\zeta(v_1) \in \mathcal{E}_0$, $\rho \models \neg stab(\zeta(v_1))$.

The waveform–completeness requirement for STD (def. 6.1) ensures that

$$exit(\zeta(v_1)) \vee stab(\zeta(v_1)) \vee enab(\zeta(v_1)) \sim \mathbf{true} \quad ;$$

since $\rho \models \neg enab(\zeta(v_1))$, this implies that $\rho \models exit(\zeta(v_1))$, which is a contradiction to (*); thus proposition (3) is proven. q.e.d.

6.4.2 Characterization of STD semantics in LTL_V^-

Recall the definition of the semantics of STD–diagrams (definition 6.15), which has been given explicitly, referring to the semantics of the POSA derived from an STD–body. The next lemma shows that the semantics of STD–diagrams can be expressed by a characteristic formula in LTL_V.

Lemma 6.2 (Temporal logic characterization of STD–diagram semantics) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and an STD–diagram $W\Delta$ over V. Then the semantics of $W\Delta$, denoted by $L(W\Delta)$, can be characterized by an equivalent LTL_V–formula $\phi_{W\Delta}$ as follows:*

- *case* $actmode(W\Delta) \equiv_{Def}$ *invariant : In this case,*

$$\phi_{W\Delta} =_{Def} \quad \mathbf{always}\,(actspec(W\Delta) \rightarrow \mathbf{next}\,\phi_{\mathcal{A}})$$

- *case* $actmode(W\Delta) \equiv_{Def}$ *initial : In this case,*

$$\phi_{W\Delta} =_{Def} \quad actexitspec(W\Delta) \vee (actspec(W\Delta) \wedge \mathbf{next}\, \phi_{\mathcal{A}})$$

where $\phi_{\mathcal{A}}$ *is the characteristic formula derived from the STD–body* $\mathcal{A} \equiv_{Def} \mathcal{A}_{W\Delta}$ *according to theorem 6.1.*

The proof of lemma 6.2 follows immediately from definition 6.15 and the definition of the semantics of the **always** –operator.

We have seen in chapter 3 that it is possible to omit the **next** –operator in LTL–specifications under certain circumstances (cf. lemma 3.14).

In terms of a POSA $\mathcal{A}$, the corresponding precondition for a Next–free characterization is that $\mathcal{A}$ is deterministic and stuttering–invariant (theorem 3.4).

Theorem 6.1 has shown that these requirements are satisfied for a POSA derived from an STD–body. Hence, the semantics of a STD–body can be characterized in LTL_V^-.

Lemma 6.3 (Characterization of STD–diagram semantics in LTL_V^-) *Assume an assertion language* $A\mathcal{L}$*, a set* V *of variables, and an STD–diagram* $W\Delta$ *over* V*. Then the semantics of* $W\Delta$*, denoted by* $L(W\Delta)$*, is characterized by the* LTL_V^-*–formula* $\phi^{\circ}_{W\Delta}$ *, i.e.*

$$\phi^{\circ}_{W\Delta} \sim \phi_{W\Delta} \quad .$$

where $\phi^{\circ}_{W\Delta}$ *is defined as follows:*

- *case* $actmode(W\Delta) =$ *invariant : In this case,*

$$\phi^{\circ}_{W\Delta} =_{Def} \quad \mathbf{always}\, (actspec(W\Delta) \rightarrow \phi^{\circ}_{\mathcal{A}})$$

- *case* $actmode(W\Delta) =$ *initial : In this case,*

$$\phi^{\circ}_{W\Delta} =_{Def} \quad actexitspec(W\Delta) \vee (actspec(W\Delta) \wedge \phi^{\circ}_{\mathcal{A}})$$

where $\phi^{\circ}_{\mathcal{A}}$ *is the characteristic formula derived from the STD–body* $\mathcal{A} \equiv_{Def} \mathcal{A}_{W\Delta}$ *according to theorem 3.4.*

Proof of lemma 6.3 . We have to consider the two possible activation modes:

- case $actmode(W\Delta) = \mathsf{invariant}$:

$$\begin{array}{rcl} \phi_{W\Delta} & =_{Def} & \mathbf{always}\,(actspec(W\Delta) \rightarrow \mathbf{next}\,\phi_{\mathcal{A}}) \\ & \sim & \mathbf{always}\,(\neg actspec(W\Delta) \vee (actspec(W\Delta) \wedge \mathbf{next}\,\phi_{\mathcal{A}})) \qquad (*) \end{array}$$

Recall that $\phi_{\mathcal{A}} \sim \phi^{\circ}_{\mathcal{A}} = \phi^{\circ}_{\zeta_0}$, where ζ_0 is the (unique) initial location of $\mathcal{A}$. According to theorem , the structure of the formula $\phi^{\circ}_{\zeta_0}$ is (with $\ell \equiv_{Def} \zeta_0$)

$$\phi^{\circ}_{\ell} =_{Def} \phi_{\ell,\ell} \quad \mathcal{U} \quad (\bigvee_{\ell \neq \ell' \,:\, \ell \rightarrow \ell'} \phi_{\ell,\ell'} \wedge \phi^{\circ}_{\ell'})$$

By lemma 3.14,

$$\begin{array}{rcl} & & actspec(W\Delta) \wedge \mathbf{next}\,\phi^{\circ}_{\zeta_0} \\ & \sim & actspec(W\Delta) \wedge \mathbf{next}\,(\phi_{\zeta_0,\zeta_0} \quad \mathcal{U} \quad (\bigvee \ldots)) \\ & \sim & [actspec(W\Delta) \Rightarrow \phi_{\zeta_0,\zeta_0}, \phi_{\zeta_0,\zeta_0} \Rightarrow \neg(\bigvee \ldots)] \\ & & actspec(W\Delta) \wedge (\phi_{\zeta_0,\zeta_0} \quad \mathcal{U} \quad (\bigvee \ldots)) \end{array}$$

hence

$$\begin{array}{rcl} (*) & \sim & \mathbf{always}\,(\neg actspec(W\Delta) \vee (actspec(W\Delta) \wedge \phi^{\circ}_{\mathcal{A}})) \\ & \equiv_{Def} & \phi^{\circ}_{W\Delta} \quad . \end{array}$$

- case $actmode(W\Delta) \equiv_{Def} \mathsf{initial}$: In this case,

$$\begin{array}{rcl} \phi^{\circ}_{W\Delta} & =_{Def} & actexitspec(W\Delta) \vee (actspec(W\Delta) \wedge \mathbf{next}\,\phi^{\circ}_{\mathcal{A}}) \\ & \sim [(**)] & actexitspec(W\Delta) \vee (actspec(W\Delta) \wedge \phi^{\circ}_{\mathcal{A}}) \\ & \sim & \phi^{\circ}_{W\Delta} \quad . \end{array}$$

where (**) is the same argument as used in the preceding case ($actmode(W\Delta) \equiv_{Def}$ invariant).

q.e.d.

6.5 Linear Decomposition

In chapter 5, we have developed a set of proof rules to reason with LSTD–specifications.

In this section, we show that the semantics of STD–diagrams can be mapped to corresponding (semantically equivalent) LSTD–specifications.

The basis for the decomposition result is theorem 5.1: For each deterministic POSA $\mathcal{A}$, there is an equivalent set ΔS of LSTD bodies, i.e. $L(\mathcal{A}) = L(\Delta S)$.

Theorem 6.2 (Linear decomposition of STD–diagrams) *Assume an assertion language $A\mathcal{L}$, a set V of variables, and an STD–diagram $W\Delta$ over V, with body $W\Delta'$.*

Then there is an equivalent LSTD–specification SPEC $\equiv_{Def}$ $SPEC_{W\Delta}$, i.e.

$$L(W\Delta) = L(SPEC)$$

Proof of theorem 6.2 – Construction. Let $\mathcal{A} \equiv_{Def} \mathcal{A}_{W\Delta'}$ be the POSA derived from the STD–body $W\Delta'$, and

$$\Delta S_{\mathcal{A}} = \{\Delta_1, \ldots, \Delta_k\}, \quad k \geq 1$$

be an equivalent set of LSTD–bodies constructed according to theorem 5.1.

We construct *SPEC* dependent of the activation mode of $W\Delta$:

- case $actmode(W\Delta) =$ invariant : Define

$$SPEC_{W\Delta} = \{\overset{\phi_1}{\vdash\!\!\!-\!\!\!-} \Delta \mid \Delta \in \Delta S\}$$

  where $\phi_1 \equiv_{Def} actspec(W\Delta)$.

- case $actmode(W\Delta) =$ initial : Define

$$SPEC_{W\Delta} = \{\overset{\phi_1}{\|\!\!\!-\!\!\!-\phi_2} \Delta \mid \Delta \in \Delta S\}$$

  where ϕ_1 is defined as above and $\phi_2 \equiv_{Def} actexitspec(W\Delta)$.

Proof of correctness of construction. We will show that the respective temporal logic characterizations are equivalent:

$$\phi(W\Delta) = \phi(SPEC) \quad .$$

- case $actmode(W\Delta) =$ invariant :

$$\begin{aligned}
\phi(SPEC_{W\Delta}) = & \bigwedge_{\Delta \in \Delta S} \mathbf{always}\,(\phi_1 \rightarrow \mathbf{next}\,\phi_\Delta) \\
\sim & \quad \mathbf{always}\,(\bigwedge_{\Delta \in \Delta S} (\phi_1 \rightarrow \mathbf{next}\,\phi_\Delta) \\
\sim & \quad \mathbf{always}\,(\phi_1 \rightarrow \mathbf{next} \bigwedge_{\Delta \in \Delta S} \phi_1 \rightarrow \mathbf{next}\,\phi_\Delta) \\
\equiv & \quad \phi(W\Delta) \quad .
\end{aligned}$$

- case $actmode(W\Delta) =$ initial :

$$\begin{aligned}
\phi(SPEC_{W\Delta}) = & \bigwedge_{\Delta \in \Delta S} ((\phi_1 \wedge \mathbf{next}\,\phi_\Delta) \vee \phi_2) \\
\sim & \quad ((\phi_1 \wedge \mathbf{next} \bigwedge_{\Delta \in \Delta S} \phi_\Delta) \vee \phi_2) \\
\equiv & \quad \phi(W\Delta) \quad .
\end{aligned}$$

q.e.d.

Example 6.6 (Linear decomposition of diagram *TL_P_and_not_Q_unless_Q*) *Consider again the diagram shown in figure 6.5 (introduced earlier in this chapter in figure 6.3) and the structure of the body–unwinding shown in figure 6.6. We show the decomposition of diagram* TL_P_and_not_Q_unless_Q *in the following in the sequence of steps:*

1. *Deriving the characteristic formula of the diagram body*
2. *Decomposition of the characteristic formula*
3. *Building the equivalent set of LSTD–diagram bodies*
4. *Adding the activation semantics to the set of LSTD–bodies, i.e. construction of an equivalent set of LSTD–diagrams.*

Step 1: Deriving the characteristic formula of the diagram body. According to theorem 6.1, we construct subformulas for the nodes labelled ζ_0 = s0, s1, and s2 = $\zeta^\top$:

Define for all locations $\ell \in Locs$

$$\phi_{\mathcal{A}} =_{Def} \phi_{s0} = (\phi_{s0,s0} \quad \textbf{unless} \quad (\bigvee_{s0 \neq s' \,:\, s0 \rightarrow s'} \phi_{s0,s'} \wedge \textbf{next}\, \phi_{s'}))$$

$$\phi_{s1} = (\phi_{s1,s1} \quad \textbf{unless} \quad (\bigvee_{s1 \neq s' \,:\, s1 \rightarrow s'} \phi_{s1,s'} \wedge \textbf{next}\, \phi_{s'}))$$

$$\phi_{s2} = (\phi_{s2,s2} \quad \textbf{unless} \quad \textbf{false})$$

where

$$\begin{aligned} \phi_{s0,s0} &= \langle\!| \text{ not(Q) and P } |\!\rangle \\ \phi_{s1,s1} &= \langle\!| \text{ P } |\!\rangle \\ \phi_{s2,s2} &= \textbf{true} \end{aligned}$$

The successor relation is: $s0 \rightarrow s1$, $s0 \rightarrow s2$, and $s1 \rightarrow s2$, and

$$\begin{aligned} \phi_{s0,s1} &= \langle\!| \text{ Q and P } |\!\rangle \\ \phi_{s0,s2} &= \langle\!| \text{ Q and not(P) } |\!\rangle \\ \phi_{s1,s2} &= \langle\!| \text{ not(P) } |\!\rangle \end{aligned}$$

Note that $\mathcal{U}$ = **unless**, because all states s0, s1, and s2 are acceptance states.

Note that the **next** –operator in the definition of ϕ_{s_i} can be omitted according to lemma 6.3.; thus the definition can be rewritten equivalently as follows:

$$\phi_{\mathcal{A}} =_{Def} \phi_{s0} = (\phi_{s0,s0} \quad \textbf{unless} \quad (\neg\phi_{s0,s0} \wedge ((\phi_{s0,s1} \wedge \phi_{s1}) \vee (\phi_{s0,s2} \wedge \phi_{s2}))))$$

$$\phi_{s1} = (\phi_{s1,s1} \quad \textbf{unless} \quad (\neg\phi_{s1,s1} \wedge (\phi_{s1,s2} \wedge \phi_{s2})))$$

$$\phi_{s2} = (\phi_{s2,s2} \quad \textbf{unless} \quad \textbf{false})$$

Step 2: Decomposition of the characteristic formula. We can next apply decomposition theorem 3.1 to the characteristic formula $\phi_{\mathcal{A}}$.

Applying this theorem to formula ϕ_{s0} defined above, we get a conjunctive decomposition of formula ϕ_{s0} into two "linear" formulas:

Figure 6.5: STD Diagram *TL_P_and_not_Q_unless_Q*.

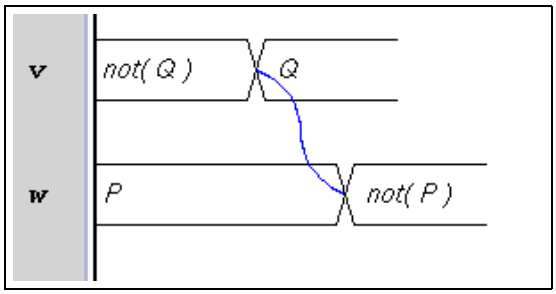
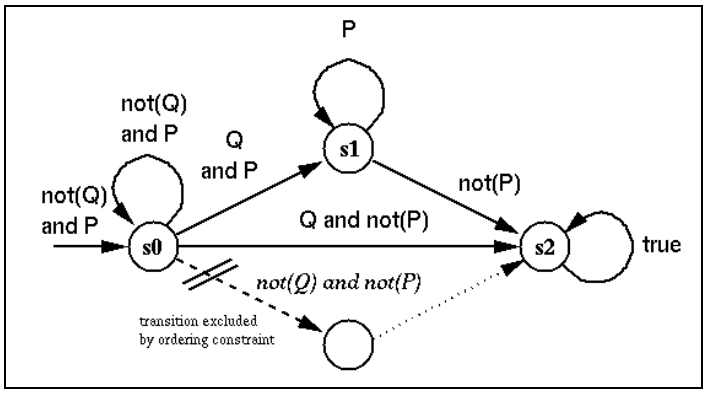


Figure 6.6: Example of POSA derived from an STD–body.

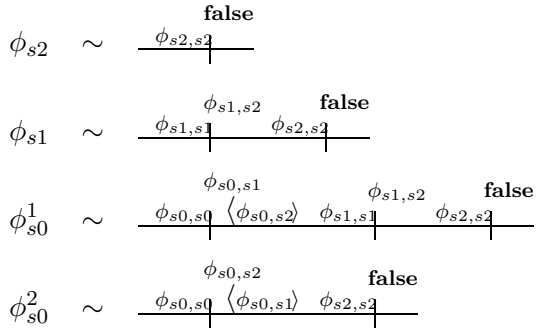


$$\phi^1_{s0} = (\phi_{s0,s0} \quad \textbf{unless} \quad (\neg\phi_{s0,s0} \wedge ((\phi_{s0,s1} \wedge \phi_{s1}) \vee (\phi_{s0,s2}))))$$
$$\phi^2_{s0} = (\phi_{s0,s0} \quad \textbf{unless} \quad (\neg\phi_{s0,s0} \wedge ((\phi_{s0,s2} \wedge \phi_{s2}) \vee (\phi_{s0,s1}))))$$

Note that formulas ϕ_{s1} and ϕ_{s2} are already in linear form.

Step 3: Building the equivalent set of LSTD–diagram bodies. Next we derive an equivalent set of LSTD–diagram bodies, which characterizes the semantics of the body of diagram *TL_P_and_not_Q_unless_Q*. The basic idea is that each **unless** / **until** –formula corresponds to the head–phase of a LSTD–body.

Therefore, we obtain the following LSTD–bodies:



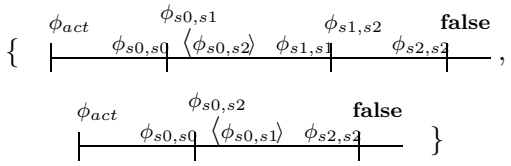
Thus, the following set of LSTD–bodies is equivalent to the semantics of the body of diagram *TL_P_and_not_Q_unless_Q*:

$$\{\phi^1_{s0}, \phi^2_{s0}\} \quad .$$

Step 4: Adding the activation semantics to the set of LSTD–bodies. The final step is the addition of the activation semantics. Diagram *TL_P_and_not_Q_unless_Q* has the activation mode Invariant, and the activation specification is:

$$actspec_{W\Delta} \equiv_{Def} \phi_{act} \equiv_{Def} \langle\!\langle \text{ not(Q) and P } \rangle\!\rangle \quad .$$

Thus, the following set of LSTD–diagrams is semantically equivalent to the diagram *TL_P_and_not_Q_unless_Q*:



q.e.d. (end example)

6.6 Summary

This chapter has introduced the definition and semantics of STD–diagrams.

STD–specifications are defined in the same way as LSTD–specifications, i.e. as sets of diagrams with a conjunctive interpretation of the diagrams in the set.

The main result of this chapter is the close connection between STD and the concept of LSTD introduced in chapter 5: The semantics of a STD diagram (and, therefore for a STD specification) can be expressed by an equivalent LSTD specification.

This allows to use the derivation rules developed for LSTD in order to establish derivation proofs between two STD diagrams $W\Delta_1$, $W\Delta_2$ as follows:

- Compute the conjunctive decompositions $SPEC_1$ and $SPEC_2$ for the corresponding STD diagrams $W\Delta_1$, $W\Delta_2$;
- Show that $SPEC_1 \Rightarrow SPEC_2$, using proof rules developed in chapter 5.

It is worth mentioning, that using linear decomposition of STD diagrams allows to perform model–checking in seperate steps, where each LSTD–diagram of the linear decomposition is verified seperately.

This can allow verification in those cases, where verification of the full formula characterizing the STD–diagram is not feasable.

Chapter 7

Resume

We have reached the end of this treatise on the development of a visual formalism for model verification.

This chapter gives a short overview of further issues, which have not been discussed in this book:

- Using STD for practical requirements capture (e.g. considerations about user interface),
- specification–pattern libraries,
- assumption/commitment style–specification,
- enhancement of expressiveness (observer specification), and
- related developments.

7.1 Using STD for practical specification

The emphasis of this work was on the theoretical foundation of STD, not on practical issues such as user interface design and the development of specification libraries.

Nevertheless, collaborate work has been performed on these topics. The first design of a user interface for STD was developed by the company Abstract Hardware (Chris Read and Colin Saunders) within the course of the FORMAT project. Later on, another implementation of a graphical editor for STD has been developed on top of Tcl/Tk by Hartmut Wittke.

7.2 Considerations about the user interface for STD

It turned out during evaluations performed by industrial project partners that subtle issues of the graphical syntax were important to the usage. For instance, the first implementation of a design tool for STD by Abstract Hardware used a graphical notation, where the precedence constraint was denoted by an arrow running "forward". The arrow was annotated with an interval $[0, \infty]$ indicating that the difference of the occurrence time of the target event was required to be greater or equal than the occurrence time of the source event.

Later on, some users found this notation confusing, because from a functional point of view, a precedence constraint really means a *past–implication*: If the target event has happened (has been matched), then the source event must have happened (be matched) at the same time or earlier. Consequently, users found that an arrow pointing "backwards" (i.e. from the target to the source event) would be more appropriate.

In this thesis, we have followed the suggestion made by Johannes Helbig in his thesis [18] to denote a precedence constraint simply by a curved line, originating to the right from the source event, and entering from the left into the target event. If this constraint is superimposed by a leadsto constraint, than a (forward pointing) curved arrow is used. A "pure" leadsto–constraint is denoted by a straight arrow.

7.2.1 The design of STDx

The actual development called STDx[1] extends the fundamental concepts of STD into several directions.

Structural extensions. As an illustration, consider the figure 7.1 showing the user–interface for the STDx–design–manager.

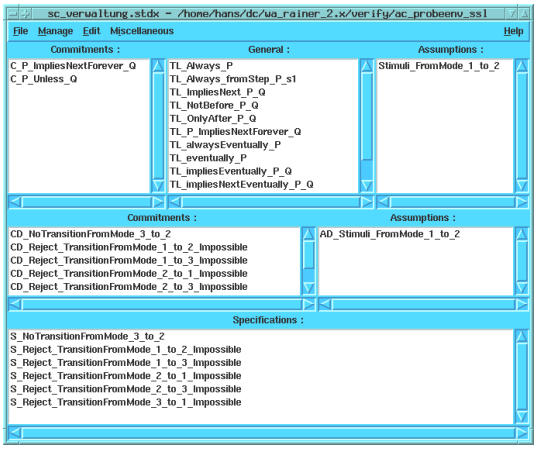
The layout of the user–interface of the STDx–specification–manager reflects the logical structure of STDx–specifications.

A specification consists of three layers:

- STDx–definitions (diagrams)
- STDx–declarations (diagram instances), and
- STDx–specification–clauses.

[1]The work on the design of STDx was partially funded by ESPRIT project no.24013 (V–FORMAT).

Figure 7.1: Snapshot of the STDx–design–manager.



A STDx–specification–clause represents an implication, which denotes a proof–obligation of the following form:

"*Prove that – provided the environment exhibits only a behaviour which is accepted by a set of* assumption*–diagrams – the system–component satisfies a specified* commitment*–diagram.*"

Furthermore, references in STDx–specifications are not made directly to diagrams, but to so–called (diagram–) *declarations*. The reason is that from a practical point of view it is useful to define diagrams in the form of (simple) *macros*, allowing parameter substitution as shown in the examples in chapter 2. An STDx–declaration is simply an instance of a STDx–definition (–diagram), where all parameters are mapped to concrete expressions.

STDx–definitions are dedicated to be used as assumptions or commitments, unless they have the type "general", which allows usage in both assumption– and commitment–declarations. General definitions are used primarily to allow the inclusion of application–specific pattern–sets. For instance, figure 7.1 shows a loaded set of definitions corresponding to basic temporal–logic idioms.

7.2.2 Guidelines for property specification using STD

The assumption/commitment–style approach to verification using STD raises a number of issues concerning safe usage. Most of these issues and observations are summerized in the poster shown in figure 7.2 (which also shows how model–

verification can be integrated into the standard methodology for a VHDL–based design flow). Some of the remarks are related to the assumption/commitment–approach, others are related specifically to the STD–method.

7.2.3 Witness–test

Although the obvious goal of verification is to obtain the result "TRUE" (meaning the proof that all possible system runs satisfy a stated property), the negative result "FALSE" is more immediately convincing. This is because the model–checker then produces a *counter–example*, which can be used to simulate the run which violates the property.

A similar idea can be used to ensure that the result "TRUE" is not trivially obtained (e.g. not caused by contradictory assumptions). It is only necessary to turn the top–frontstate into a non–accepting state. The consequence is that the model–checker searches for a path, which (a) activates the diagram describing the property, and (b) shows a complete matching sequence which leads to the top–frontstate of that diagram. Then, the error–path stops, because no further continuation is possible.

This is called the "*witness–test*". E.g., in figure 7.1 several specification clauses are shown. Some of these are used to demonstrate the impossibility of transitions between certain control–modes of an embedded controllers, while others exploit the technique of the witness–test to prove the opposite case, i.e. the *existence* (or possibility) of certain control–mode transitions in the controller.

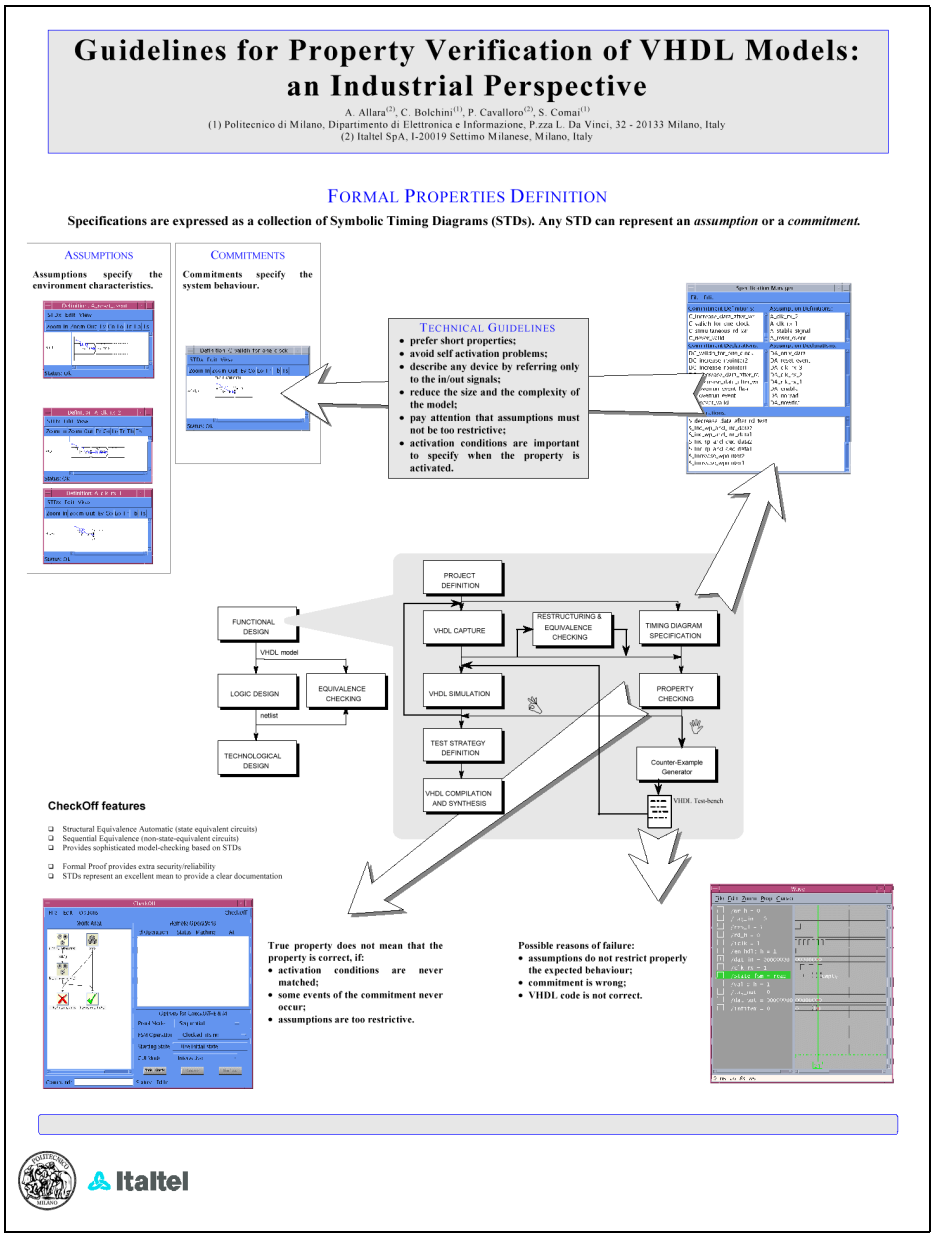
7.3 Enhancement of expressiveness of STD

The expressiveness of STD can be extended into various directions, including:

- support of (synchronous) real–time (RTSTD, STDx), and
- support of specification variables.

Timing constraints. STDx supports the possibility to express properties using quantiative timing constraints. A quantitative timing constraint is a precedence constraint with an annotated timing interval $[m, n]$, where m and n are constant natural numbers, $m \leq n$. The semantics is similar to the development of RTSTD ([14]); the main difference is the definition of constraint–priority (violation of weak constraints have priority over the violation of strong constraints) and the restriction that quantitative–timing constraints in STDx

Figure 7.2: Guidelines for property verification of VHDL models.



refer to a discrete clock. Thus, it is possible to describe the semantics of STDx using the nexttime–operator of LTL. This ensures still an efficient verification complexity, provided the numbers occuring in constraint intervals are reasonably small.

Specification variables. Another important extension is the possibility to define and refer to *specification variables* inside STDx–specifications. There are two types of specifications variables:

- *rigid specification variables* are used to express universal quantification, typically over variables carrying multi–valued data (e.g. array–indices). This technique is described in detail in the standard text [26].
- *flexible specification variables* behave like additional model–inputs. They can be used as a memory for the specification.

The addition of flexible specification variable increases the expressive power of the STD–formalism and also allows to introduce structure into a specification.

Typically the behaviour of specification variables is controlled by so–called *observer specifications*. An observer is an automaton, which monitors the logical state of an interface dependent on the history of events. An observer can most naturally be described by a finite state machine, but it is also possible to use STD–assumptions to define an observer. The report [1] describes this approach in more detail.

7.4 Related developments

Recently, an extension of message–sequence–charts named life–sequence–charts (LSC's) has been developed by W.Damm and D.Harel [12]. The approach of LSC is complementary to STD due to the focus on the *interaction* between components (while in STD the focus is on the interface of an isolated system–component).

LSC extends some ideas from STD, e.g. "hot" and "cold" conditions corresponding to instable respectively stable frontstates in STD diagrams, the concept of activation, and the concept of an exit from a chart.

LSC is also intended to be used for property specification. Although this has not been formally developed yet, it is likely that (at least a close variant of the current definition) will subsume the possibilities provided by the STD formalism.

A big advantage of LSC is that it adheres to the widely used standard definition of MSC's (which is not the case for STD).

Appendix A

Proofs

A.1 Proof of theorem 3.3

We present here the remaining part of the proof of theorem 3.3 (see page 64)

Proof of theorem 3.3 , continued. We continue the proof of theorem 3.3 by induction; show that

$$\forall i \geq 0 \; \forall \ell \in T_i \; : \; L_\ell(\mathcal{A}) = L(\phi\chi_\ell)$$

where

$$\begin{array}{rcl} L_\ell(\mathcal{A}) & \equiv_{Def} & L(\mathcal{A}^\ell) \quad \text{, and} \\ \mathcal{A}^\ell & \equiv_{Def} & (V, \mathit{Locs}, \mathit{Edges}, \{\ell\}, F) \end{array}$$

is the same SA as $\mathcal{A}$, except for the changed set $\{\ell\}$ of initial locations. In particular,

$$\sigma \in L_\ell(\mathcal{A}) \quad \textbf{iff} \quad \begin{array}{l} \exists \sigma\ell \; . \; \sigma\ell \text{ is an accepting run} \\ \text{over } \sigma \text{ in } \mathcal{A} \text{ starting from location } \ell \end{array}$$

case $i = 0$ (Induction base case). Show that $\forall \ell \in T_0 \; : \; L_\ell(\mathcal{A}) = L(\phi\chi_\ell) \quad (*)$.

case (i). Consider arbitrary fixed $\ell \in T_0 \cap F$; show $(*)$.
$\subseteq$: Assume arbitrary fixed $\sigma \in L_\ell(\mathcal{A})$, $\sigma \equiv_{Def} (\rho_i)_{i \geq 0}$.
By definition, $\ell \in T_0$,
$\hookrightarrow$[prop.(3): $T = T_0$] $\ell \in T$.

$\hookrightarrow$[def. of T] $\neg\exists \ell' \neq \ell \,.\, \ell \to \ell'$ $\hookrightarrow[\sigma \in L_\ell(\mathcal{A})]$

$\exists$ accepting run $\sigma\ell$ over σ in $\mathcal{A}$ starting from location ℓ

$\hookrightarrow[\neg\exists \ell' \neq \ell \,.\, \ell \to \ell']$ $\sigma\ell = \ell\,\ell\ldots \wedge \forall i \geq 0 \,:\, \rho_i \models \phi_{\ell,\ell}$

$\hookrightarrow$[def. of $\phi\chi_\ell$, $\ell \in F$, $\bigvee_{\emptyset} \phi \sim$ **false**] $\phi\chi_\ell = \phi_{\ell,\ell}$ **unless false**

$\hookrightarrow$[def. of **unless**] $\sigma \in L(\phi\chi_\ell)$.

$\supseteq$: Assume that $\sigma \in L(\phi\chi_\ell)$, $\sigma \equiv_{Def} (\rho_i)_{i\geq 0}$. Again, $\ell \in T_0 = T$

$\hookrightarrow$[def. of T] $\neg\exists \ell' \neq \ell \,.\, \ell \to \ell'$

$\hookrightarrow$[def. of $\phi\chi_\ell$, $\ell \in F$, $\bigvee_{\emptyset} \phi \sim$ **false**] $\phi\chi_\ell = \phi_{\ell,\ell}$ **unless false**

$\hookrightarrow[\sigma \in L(\phi\chi_\ell)]$ $\sigma \models \phi_{\ell,\ell}$ **unless false**

$\hookrightarrow[\phi$ **unless false** $\sim$ **always** $\phi]$ $\forall i \geq 0 \,:\, \rho_i \models \phi_{\ell,\ell}$

$\hookrightarrow$[with $\sigma\ell = \ell\,\ell\ldots$, and $\forall i \geq 0 \,:\, \rho_i \models \phi_{\ell,\ell}$]

$\exists$ accepting run $\sigma\ell$ over σ in $\mathcal{A}$ starting from location ℓ

$\hookrightarrow$[def. of $L_\ell(\mathcal{A})$] $\sigma \in L_\ell(\mathcal{A})$.

case (ii) . Consider arbitrary fixed $\ell \in T_0 \backslash F$; show $L_\ell(\mathcal{A}) = \emptyset = L(\phi\chi_\ell)$, from which in particular $(*)$ follows. $L_\ell(\mathcal{A}) = \emptyset$ Assume that some $\sigma \in L_\ell(\mathcal{A})$ $(\bar{\#})$ exists. The only possible run starting from ℓ is $\sigma\ell = \ell\,\ell\ldots$; but since $\ell \notin F$, $\sigma\ell$ is not accepting and $(\bar{\#})$ cannot be true.

Hence, $L_\ell(\mathcal{A}) = \emptyset$. $\emptyset = L(\phi\chi_\ell)$ By premise, $\ell \notin F$,

$\hookrightarrow$[def. of $\phi\chi_\ell$, $\ell \notin F$, $\bigvee_{\emptyset} \phi \sim$ **false**] $\phi\chi_\ell = \phi_{\ell,\ell}$ **until false**

$\hookrightarrow[\phi$ **until false** $\sim$ **false**] $L(\phi\chi_\ell) = \emptyset$.

case $i \to i+1$ **(Induction step).** Assume that for arbitrary fixed i,
$\forall \ell \in T_i \,:\, L_\ell(\mathcal{A}) = L(\phi\chi_\ell)$ $(*)$.
Let $\forall i \geq 0 \,:\, \partial T_{i+1} =_{Def} T_{i+1} \backslash T_i$. It suffices to show that
$\forall \ell \in \partial T_{i+1} \,:\, L_\ell(\mathcal{A}) = L(\phi\chi_\ell)$ $(**)$

$\hookrightarrow[T_{i+1} = \partial T_{i+1} \cup T_i, (*)]$ $\forall \ell \in T_{i+1} \,:\, L_\ell(\mathcal{A}) = L(\phi\chi_\ell)$.

case (i). Consider arbitrary fixed $\ell \in \partial T_{i+1} \cap F$; show $(**)$. $\subseteq$ Let $\sigma \in L_\ell(\mathcal{A})$, $\sigma \equiv_{Def} (\rho_i)_{i\geq 0}$.

$\hookrightarrow$[def. of $L_\ell(\mathcal{A})$]

$\exists$ accepting run $\sigma\ell$ over σ in $\mathcal{A}$ starting from location ℓ, $\ell \in F$.

case 1. $\sigma\ell = \ell\,\ell\ldots$

$\hookrightarrow$[$\sigma\ell$ is a run over σ] $\forall i \geq 0 \;:\; \rho_i \models \phi_{\ell,\ell}$

$\hookrightarrow$[def. of **always** $\phi_{\ell,\ell}$] $\sigma \models \mathbf{always}\,\phi_{\ell,\ell}$

$\hookrightarrow$[$\forall \phi' \;:\; \mathbf{always}\,\phi \Rightarrow \phi\,\mathbf{unless}\,\phi'$, $\ell \in F$,

$\phi\chi_\ell = \phi_{\ell,\ell}\,\mathbf{unless}\,\phi'$ for some ϕ']

$\sigma \models \phi\chi_\ell$

$\hookrightarrow$[def. of $L(\phi\chi_\ell)$] $\sigma \in L(\phi\chi_\ell)$.

case 2. $\sigma\ell \equiv_{Def} (\ell_i)_{i\geq 0}$ has the form

$\sigma\ell = \underbrace{\ell\ldots\ell}_{k\geq 1}\,\underbrace{\ell'\ldots}_{\sigma\ell^{(k)}}$

for some $k \geq 1$, where $\ell \neq \ell'$, and $\ell \to \ell'$.

$\hookrightarrow$[$\ell \in T_{i+1}$, lemma 3.12, property (5)] $\ell' \in T_i$

$\hookrightarrow$[with $\phi_j \equiv_{Def} \phi_{\ell_j,\ell_{j+1}}$] $\forall j \geq 0 \;:\; (\ell_j, \phi_j, \ell_{j+1}) \in \mathit{Edges} \wedge \rho_j \models \phi_j$

$\hookrightarrow$[$\sigma\ell = \underbrace{\ell\ldots\ell}_{k\geq 1}\,\ell'\ldots \quad\Longrightarrow\quad \forall j \,.\, 0 \leq j \leq (k-1) : \ell_j = \ell \wedge \ell_k = \ell'$]

$(<)$	$\forall j \,.\, 0 \leq j < (k-1)\;:$	$\rho_j \models \phi_{\ell,\ell}$
$(=)$	$\wedge\ [j = k-1]$	$\rho_{k-1} \models \phi_{\ell,\ell'}$
$(>)$	$\wedge$	$\sigma\ell^{(k)}$ is a run in $\mathcal{A}$ over $\sigma^{(k)}$ starting from location ℓ'

Our next goal is to show that

$$\sigma \models \phi_{\ell,\ell}\,\mathbf{unless}\,(\phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'}) \quad (\#)$$

$\hookrightarrow$[$\forall \phi, \phi_1, \phi_2 \;:\; \phi_1 \Rightarrow \phi_2 \quad\Longrightarrow\quad \phi\,\mathbf{unless}\,\phi_1 \Rightarrow \phi\,\mathbf{unless}\,\phi_2$]

$\sigma \models \phi_{\ell,\ell}\,\mathbf{unless}\,(\bigvee_{\ell\neq\ell' \;:\; \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'})$

$\hookrightarrow$[def. of $\phi\chi_\ell$] $\sigma \models \phi\chi_\ell$

$\hookrightarrow$[def. of $L(\phi\chi_\ell)$] $\sigma \in L(\phi\chi_\ell)$.

So, it remains to show (#), under the premise that the clauses marked $(<)$, $(=)$ and $(>)$ hold.

$\hookrightarrow$[$(<)$ and $(=)$, $k' \equiv_{Def} k-1$, $k \geq 1$]

$\exists k' \geq 0 \,.\, \sigma^{(k')} \models \phi_{\ell,\ell'} \wedge \forall j \,.\, 0 \leq j < k' \;:\; \sigma^{(j)} \models \phi_{\ell,\ell}$

From $(>)$ follows that $\sigma\ell^{(k)}$ is an *accepting* run over $\sigma^{(k)}$ in $\mathcal{A}$ starting from location ℓ', since each location which occurs infinitely often in $\sigma\ell$, does also occur infinitely often in $\sigma\ell^{(k)}$.

Hence, if $\sigma\ell$ is accepting, then so is $\sigma\ell^{(k)}$.

$\hookrightarrow$[$\ell' \in T_i$] $\sigma^{(k)} \in L_{\ell'}(\mathcal{A})$

$\hookrightarrow$[Induction hypothesis] $\sigma^{(k)} \models \phi\chi_{\ell'}$

$\hookrightarrow$[def. of **next**] $\sigma^{(k-1)} \models \mathbf{next}\,\phi\chi_{\ell'}$.

$\hookrightarrow$[summarizing the above argument, $k' \equiv_{Def} k-1$]

$\exists k' \geq 0 \,.\, \sigma^{(k')} \models \phi_{\ell,\ell'} \wedge \sigma^{(k')} \models \mathbf{next}\,\phi\chi_{\ell'} \wedge \forall j \,.\, 0 \leq j < k' \;:\; \sigma^{(j)} \models \phi_{\ell,\ell}$

$\hookrightarrow$[def. of **unless**] $(\#)$.

$\supseteq$ Let $\sigma \in L(\phi\chi_\ell)$, $\ell \in F$.

$\hookrightarrow$[def. of $L(\phi\chi_\ell)$] $\sigma \models \phi\chi_\ell$

$\hookrightarrow$[def. of $\phi\chi_\ell$] $\sigma \models \phi_{\ell,\ell}\,\mathbf{unless}\,(\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'})$

$\hookrightarrow$[def. of **unless**] $\sigma \models \mathbf{always}\,\phi_{\ell,\ell}$

$\quad\vee\quad \sigma \models \phi_{\ell,\ell}\,\mathbf{until}\,(\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'})$

case 1. $\sigma \models \mathbf{always}\,\phi_{\ell,\ell}$

$\hookrightarrow$[def. of **always**] $\forall i \geq 0 \;:\; \rho_i \models \phi_{\ell,\ell}$

$\hookrightarrow$[$\sigma\ell \equiv_{Def} \ell\,\ell\ldots, \ell \in F$]

$\exists \sigma\ell \,.\, \sigma\ell$ is an accepting run in $\mathcal{A}$ over σ starting from location ℓ

$\hookrightarrow$[def. of $L_\ell(\mathcal{A})$] $\sigma \in L_\ell(\mathcal{A})$.

case 2. $\sigma \models \phi_{\ell,\ell}\,\mathbf{until}\,(\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'})$

$\hookrightarrow$[def. of **until**] $\exists k' \geq 0 \,.\, \sigma^{(k')} \models (\bigvee_{\ell\neq\ell' \,:\, \ell\to\ell'} \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'})$

$\quad\wedge\quad \forall j \,.\, 0 \leq j < k' \;:\; \sigma^{(j)} \models \phi_{\ell,\ell}$

$\hookrightarrow$[def. of $\bigvee$] $\exists k' \geq 0\; \exists \ell' \,.\, \ell \neq \ell' \wedge \ell \to \ell' \wedge$

$\qquad \sigma^{(k')} \models \phi_{\ell,\ell'} \wedge \mathbf{next}\,\phi\chi_{\ell'}$

$\quad\wedge\quad \forall j \,.\, 0 \leq j < k' \;:\; \sigma^{(j)} \models \phi_{\ell,\ell}$

$\hookrightarrow$[$k' \equiv_{Def} k-1$, def. of **next**, $\sigma^{(k-1)^{(1)}} = \sigma^{(k-1+1)}$]

$\exists k \geq 1\; \exists \ell' \,.\, \ell \neq \ell' \wedge \ell \to \ell' \wedge$

$\qquad \sigma^{(k-1)} \models \phi_{\ell,\ell'} \wedge \sigma^{(k)} \models \phi\chi_{\ell'}$

$\quad \wedge \quad \forall j \,.\, 0 \le j < (k-1) \;:\; \sigma^{(j)} \models \phi_{\ell,\ell}$

$\hookrightarrow$[$\ell' \in T_i$ and $\sigma^{(k)} \in L(\phi\chi_{\ell'})$, so by induction hypothesis: $\sigma^{(k)} \in L_{\ell'}(\mathcal{A})$]

$\exists k \ge 1 \; \exists \ell' \,.\, \ell \ne \ell' \wedge \ell \to \ell' \wedge$

(1)	$\forall j \,.\, 0 \le j < (k-1) \;:$	$\rho_j \models \phi_{\ell,\ell}$
(2)	$\wedge$	$\rho_{k-1} \models \phi_{\ell,\ell'}$
(3)	$\wedge$	$\exists \sigma\ell' \,.\, \sigma\ell'$ is an accepting run in $\mathcal{A}$ over $\sigma^{(k)}$ starting from location ℓ'

$\hookrightarrow$[if $\sigma\ell'$ is accepting, then so is $\sigma\ell \equiv_{Def} \underbrace{\ell \ldots \ell}_{k \ge 1} \sigma\ell'$, for all $k \ge 1$]

$\exists \sigma\ell = \underbrace{\ell \ldots \ell}_{k \ge 1} \underbrace{\ell' \ldots}_{\sigma\ell'}$ such that

$\sigma\ell$ is an accepting run in $\mathcal{A}$ over σ starting from location ℓ

$\hookrightarrow$[def. of $L_\ell(\mathcal{A})$] $\sigma \in L_\ell(\mathcal{A})$.

case (ii). Consider arbitrary fixed $\ell \in \partial T_{i+1} \backslash F$; show $(**)$.

This case is a special case of (i). Since $\ell \notin F$, in both directions $\subseteq$ and $\supseteq$ case 1 does not exist; the argument chains of the directions $\subseteq$ and $\supseteq$ in case 2 can be transcribed easily replacing the operator **unless** by the operator **until** at the obvious places. Hence, $(**)$ holds.

Proof of $*\chi$. So far, we have established that

$$\forall \ell \in R \;:\; L_\ell(\mathcal{A}) = L(\phi\chi_\ell) \quad (***).$$

It remains to show that

$$L(\mathcal{A}) = \bigcup_{\ell \in L_0} L_\ell(\mathcal{A}) \quad (\#\#)$$

since for all $\sigma \in Comp(V)$,

$\sigma \in L(\mathcal{A})$ $\Leftrightarrow$[def. of $L(\mathcal{A})$,$(\#\#)$] $\exists \ell_0 \in L_0 \,.\, \sigma \in L_{\ell_0}(\mathcal{A})$

$\Leftrightarrow$[$L_0 \subseteq R$, $(***)$] $\exists \ell_0 \in L_0 \,.\, \sigma \in L(\phi\chi_{\ell_0})$

$\Leftrightarrow$[def. of $\phi_{\mathcal{A}}$] $\sigma \in L(\phi_{\mathcal{A}})$

hence $*\chi$ follows from $(\#\#)$.

For all $\sigma \in Comp(V)$,

$\sigma \in L(\mathcal{A})$

$\Leftrightarrow$[def. of $L(\mathcal{A})$] $\exists \ell_0 \in L_0$, $\exists$ accepting run in $\mathcal{A}$ over σ starting from ℓ_0

$\Leftrightarrow$[def. of $L_{\ell_0}(\mathcal{A})$] $\exists \ell_0 \in L_0 \,.\, \sigma \in L_{\ell_0}(\mathcal{A})$

$\Leftrightarrow$[def. of $\bigcup$] $\sigma \in \bigcup_{\ell \in L_0} L_\ell(\mathcal{A})$.

hence (##) follows and the proof of theorem 3.3 is complete.

A.2 Proof of lemma 3.14

Proof of lemma 3.14 , (1). Consider the case $\mathcal{U} \equiv_{is}$ **unless**; let $\phi \equiv_{Def} \phi_{\,\mathbf{unless}}$ and $\phi^\circ \equiv_{Def} \phi^\circ_{\mathbf{unless}}$. We have to show that

$$\forall \sigma \in Comp(V) \ : \ \sigma \models \phi \rightarrow \sigma \models \phi^\circ.$$

Assume that for arbitrary fixed σ, $\sigma \models \phi$ holds, i.e.

$$\sigma \models \phi_0 \ (*) \text{ and } \sigma \models \mathbf{next}\,(\phi_1 \,\mathbf{unless}\, \phi_2)$$

$\hookrightarrow$[def. of **next**] $\sigma^{(1)} \models \phi_1 \,\mathbf{unless}\, \phi_2$.

- Case 1: $\sigma^{(1)} \models \mathbf{always}\,\phi_1$

$\hookrightarrow$[def. of **always**] $\forall k \geq 0 \ : \ \sigma^{(1)^{(k)}} \models \phi_1$

$\hookrightarrow$[$\sigma^{(1)^{(k)}} = \sigma^{(1+k)}$] $\forall k \geq 0 \ : \ \sigma^{(1+k)} \models \phi_1$

$\hookrightarrow$[$k' \equiv_{Def} (k+1)$] $\forall k' \,.\, k'-1 \geq 0 \ : \ \sigma^{(k')} \models \phi_1$

$\hookrightarrow$[$k'-1 \geq 0$ **iff** $k' \geq 1$] $\forall k' \geq 1 \ : \ \sigma^{(k')} \models \phi_1$

$\hookrightarrow$[(*): $\sigma \models \phi_0$; since $\phi_0 \Rightarrow \phi_1$, $\sigma \models \phi_1$; $\sigma = \sigma^{(0)}$]
$\sigma^{(0)} \models \phi_1$ and $\forall k' \geq 1 \ : \ \sigma^{(k')} \models \phi_1$

$\hookrightarrow$[$k' \geq 0$ **iff** $k' \geq 1 \vee k' = 0$] $\forall k' \geq 0 \ : \ \sigma^{(k')} \models \phi_1$

$\hookrightarrow$[def. of **always**] $\sigma \models \mathbf{always}\,\phi_1$;

$\hookrightarrow$[def. of **unless**] $\sigma \models \phi_1 \,\mathbf{unless}\, \phi_2$

$\hookrightarrow$[with (*)] $\sigma \models \phi^\circ$.

- Case 2: $\sigma^{(1)} \not\models \mathbf{always}\,\phi_1$

$\hookrightarrow$[def. of **unless**] $\exists k \geq 0 \,.\, \sigma^{(1)^{(k)}} \models \phi_2$
$\wedge\ \forall j \,.\, 0 \leq j < k \ : \ \sigma^{(1)^{(j)}} \models \phi_1$

$\hookrightarrow$[$\sigma^{(1)^{(k)}} = \sigma^{(1+k)}$] $\exists k \geq 0 \ : \ \sigma^{(1+k)} \models \phi_2$
$\wedge\ \forall j \,.\, 0 \leq j < k \ : \ \sigma^{(1+j)} \models \phi_1$

$\hookrightarrow$[$k' \equiv_{Def} (k+1)$] $\exists k' \,.\, k'-1 \geq 0 \wedge \sigma^{(k')} \models \phi_2$
$\wedge\ \forall j \,.\, 0 \leq j < (k'-1) \ : \ \sigma^{(1+j)} \models \phi_1$

$\hookrightarrow$[$j' \equiv_{Def} (j+1)$] $\exists k' \,.\, k'-1 \geq 0 \wedge \sigma^{(k')} \models \phi_2$

$\wedge\ \forall j' \,.\, 0 \leq (j'-1) < (k'-1)\ :\ \sigma^{(j')} \models \phi_1$

$\hookrightarrow$[$k'-1 \geq 0$ **iff** $k' \geq 1, 0 \leq (j'-1) < (k'-1)$ **iff** $1 \leq j' < k'$]

$\exists k' \,.\, k' \geq 1 \wedge \sigma^{(k')} \models \phi_2$
$\wedge\ \forall j' \,.\, 1 \leq j' < k'\ :\ \sigma^{(j')} \models \phi_1$

$\hookrightarrow$[(*): $\sigma \models \phi_0$; since $\phi_0 \Rightarrow \phi_1$, $\sigma \models \phi_1$; $\sigma = \sigma^{(0)}$]

$\exists k' \,.\, k'-1 \geq 0 \wedge \sigma^{(k')} \models \phi_2$
$\wedge\ \forall j' \,.\, 1 \leq j' < k'\ :\ \sigma^{(j')} \models \phi_1$
$\wedge\ \sigma^{(0)} \models \phi_1$

$\hookrightarrow$[for $k' \geq 1$: $0 \leq j' < k'$ **iff** $1 \leq j' < k' \vee 0 = j'$]

$\exists k' \,.\, k'-1 \geq 0 \wedge \sigma^{(k')} \models \phi_2$
$\wedge\ \forall j' \,.\, 0 \leq j' < k'\ :\ \sigma^{(j')} \models \phi_1$

$\hookrightarrow$[$\exists k' \geq 1 \,.\, P(k') \Rightarrow \exists k' \geq 0 \,.\, P(k')$, def. of **unless**] $\sigma \models \phi_1$ **unless** ϕ_2

$\hookrightarrow$[(*)] $\sigma \models \phi^\circ$.

Second, consider the case $\mathcal{U} \equiv_{is}$ **until**; let $\phi \equiv_{Def} \phi_{\,\textbf{until}}$ and $\phi^\circ \equiv_{Def} \phi^\circ_{\textbf{until}}$. The proof runs as shown above in case 2; note that case 1 cannot occur.

q.e.d.

Proof of lemma 3.14 , (2). Consider first the case $\mathcal{U} \equiv_{is}$ **unless**; let $\phi \equiv_{Def} \phi_{\,\textbf{unless}}$ and $\phi^\circ \equiv_{Def} \phi^\circ_{\textbf{unless}}$. It suffices to show that under the premise

$$\phi_0 \Rightarrow \phi_1 \wedge \phi_1 \Rightarrow \neg\phi_2 \quad (**)$$

it follows that $\phi^\circ \Rightarrow \phi$ (+). Under the premise (**), $\phi \Rightarrow \phi^\circ$ holds by property (1), hence (2) follows from (1) and (+).

So we have to show (+), i.e.

$$\forall \sigma \in \mathit{Comp}(V)\ :\ \sigma \models \phi^\circ \rightarrow \sigma \models \phi \quad .$$

Assume that for arbitrary fixed σ, $\sigma \models \phi^\circ$ holds, i.e. $\sigma \models \phi_0 (*+)$ and $\sigma \models (\phi_1$ **unless** $\phi_2)$.

- Case 1: $\sigma \models$ **always** ϕ_1;

$\hookrightarrow$[def. of **always**] $\forall k' \geq 0\ :\ \sigma^{(k')} \models \phi_1$

$\hookrightarrow$[in particular] $\forall k' \geq 1\ :\ \sigma^{(1+(k'-1))} \models \phi_1$

$\hookrightarrow$[$k \equiv_{Def} (k'-1)$] $\forall k \,.\, k+1 \geq 1 \;:\; \sigma^{(1+k)} \models \phi_1$

$\hookrightarrow$[$k+1 \geq 1$ **iff** $k \geq 0$, $\sigma^{(1+k)} = \sigma^{(1)^{(k)}}$] $\forall k \geq 0 \;:\; \sigma^{(1)^{(k)}} \models \phi_1$

$\hookrightarrow$[def. of **always**] $\sigma^{(1)} \models$ **always** ϕ_1

$\hookrightarrow$[def. of **unless**] $\sigma^{(1)} \models \phi_1$ **unless** ϕ_2

$\hookrightarrow$[def. of **next**] $\sigma \models$ **next** $(\phi_1$ **unless** $\phi_2)$

$\hookrightarrow$[def. of ϕ, $(*+)$] $\sigma \models \phi$.

- Case 2: $\sigma \not\models$ **always** ϕ_1

$\hookrightarrow$[def. of **unless**] $\exists k' \geq 0 \,.\, \sigma^{(k')} \models \phi_2$
$\wedge\ \forall j' \,.\, 0 \leq j' < k' \;:\; \sigma^{(j')} \models \phi_1$

$\hookrightarrow$[$k' \geq 0$ **iff** $k' = 0 \vee k' \geq 1$] $\sigma^{(0)} \models \phi_2 \vee$
$\exists k' \geq 1 \,.\, \sigma^{(k')} \models \phi_2$
$\wedge\ \forall j' \,.\, 0 \leq j' < k' \;:\; \sigma^{(j')} \models \phi_1$

$\hookrightarrow$[By premise $(**)$, $\phi_0 \Rightarrow \phi_1 \wedge \phi_1 \Rightarrow \neg\phi_2$;
so from $(*+)$ $(\sigma \models \phi_0)$ follows $\sigma \models \neg\phi_2$]
$\exists k' \geq 1 \,.\, \sigma^{(1+(k'-1))} \models \phi_2$
$\wedge\ \forall j' \,.\, 1 \leq j' < k' \;:\; \sigma^{(1+(j'-1))} \models \phi_1$

$\hookrightarrow$[$k \equiv_{Def} k'-1$, $j \equiv_{Def} j'-1$]
$\exists k \,.\, k+1 \geq 1 \,.\, \sigma^{(1+k)} \models \phi_2$
$\wedge\ \forall j \,.\, 1 \leq (j+1) < (k+1) \;:\; \sigma^{(1+j)} \models \phi_1$

$\hookrightarrow$[$k+1 \geq 1$ **iff** $k \geq 0$, $1 \leq (j+1) < (k+1)$ **iff** $0 \leq j < k$]
$\exists k \,.\, k \geq 0 \,.\, \sigma^{(1+k)} \models \phi_2$
$\wedge\ \forall j \,.\, 0 \leq j < k \;:\; \sigma^{(1+j)} \models \phi_1$

$\hookrightarrow$[$\sigma^{(1+k)} = \sigma^{(1)^{(k)}}$]
$\exists k \,.\, k \geq 0 \,.\, \sigma^{(1)^{(k)}} \models \phi_2 \wedge \forall j \,.\, 0 \leq j < k \;:\; \sigma^{(1)^{(j)}} \models \phi_1$

$\hookrightarrow$[def. of **unless**] $\sigma^{(1)} \models \phi_1$ **unless** ϕ_2

$\hookrightarrow$[def. of **next**] $\sigma \models$ **next** $(\phi_1$ **unless** $\phi_2)$

$\hookrightarrow$[def. of ϕ, $(*+)$] $\sigma \models \phi$.

Second, consider the case $\mathcal{U} \equiv_{is}$ **until**; let $\phi^\circ \equiv_{Def} \phi^\circ_{\textbf{until}}$ and $\phi \equiv_{Def} \phi_{\,\textbf{until}}$. The proof runs as shown above in case 2; note that case 1 cannot occur.

q.e.d.

A.3 Proof of theorem 4.1

Recall the claim of theorem 4.1:

Assume a set V of variables and an assertion language $A\mathcal{L}$, OTGS $\mathcal{G}M_1$, $\mathcal{G}M_2$, and formulas ϕ_1, ϕ_2 over $V_i \equiv_{Def} V_{\mathcal{G}M_i}$ (for $i = 1, 2$); let $\mathcal{G}M$ be a composition of $\mathcal{G}M_1$ and $\mathcal{G}M_2$, $V \equiv_{Def} V_{\mathcal{G}M}$.

Then

$$\mathcal{G}M_1 \models \phi_1 \textbf{ and } \mathcal{G}M_2 \models \phi_2 \quad \Longrightarrow \quad \mathcal{G}M \models \phi_1 \wedge \phi_2 \quad .$$

Proof outline. Assume that $\mathcal{G}M_1 \models \phi_1$ **and** $\mathcal{G}M_2 \models \phi_2$ (*), i.e.
$\forall \sigma \in L(\mathcal{G}M_1) \ : \ \sigma \models \phi_1$, and $\forall \sigma \in L(\mathcal{G}M_2) \ : \ \sigma \models \phi_2$.

Show that

$$\forall \sigma \in L(\mathcal{G}M) \ : \ \sigma \models \phi_1 \wedge \phi_2 \qquad (\#)$$

Assume that (#) does not hold, i.e.

$$\exists \sigma_0 \in L(\mathcal{G}M) \, . \, \sigma_0 \not\models \phi_1 \wedge \phi_2 \qquad (\bar{\#})$$

W.l.o.g we may assume that $\sigma_0 \not\models \phi_1$;

$\hookrightarrow$[$\sigma_0 \in L(\mathcal{G}M)$] $\sigma_0 \in L(TS_{\mathcal{G}M})$.

σ_0 is a computation over V, i.e. a sequence $(\rho_i)_{i \geq 0}$ of states over V.

By definition, $\sigma_0 \in L(TS_{\mathcal{G}M})$ implies (with $TS =_{Def} TS_{\mathcal{G}M}$)

$\hookrightarrow$[Initiation] $[\![\Theta_{TS}]\!]\rho_0 = \textbf{true}$ (**); and

$\hookrightarrow$[Consecution] σ_0 is justified by some transition sequence

$\sigma\tau =_{Def} (\tau_i)_{i \geq 0}$ of transitions contained in $\mathcal{T} \equiv_{Def} \mathcal{T}_{\mathcal{G}M}$, i.e.

$$\forall i \geq 0 \ : \ \rho_{i+1} \in \tau_i(\rho_i) \qquad (***)$$

Our next proof goal (goal 1) is to show that under assumption ($\bar{\#}$) we may conclude that

$$\sigma_1 \equiv_{Def} \sigma_{0|V_1} \in L(\mathcal{G}M_1) \qquad (\tilde{*})$$

Since $V_1 \subseteq V$ and *free*$(\phi_1) \subseteq V_1$, lemma 4.1,(2) applies; in particular

$\hookrightarrow$[$\sigma_0 \not\models \phi_1$] $\sigma_1 \not\models \phi_1$

which is together with ($\tilde{*}$) a contradiction to premise (*); hence we will have shown that (#) follows from (*).

Proof of goal 1 . We have to show $(\tilde{*})$; let $\sigma_1 =_{Def} (\tilde{\rho}_i)_{i\geq 0}$. First show that

$$[\![\Theta_{TS_1}]\!]\tilde{\rho}_0 = \mathbf{true} \textbf{ (Initiation)}$$

where $TS_1 \equiv_{Def} TS_{\mathcal{G}M_1}$, $\tilde{\rho}_0 = \rho_{0|V_1}$.

It can easily be seen from the construction of Θ_{TS}, that Θ_{TS_1} has less **and**–conjuncts than Θ_{TS}

$\hookrightarrow$[$[\![P \textbf{ and } Q]\!]\rho = \mathbf{true} \Rightarrow [\![P]\!]\rho = \mathbf{true}, (**)$] $[\![\Theta_{TS_1}]\!]\rho_0 = \mathbf{true}$.

$\hookrightarrow$[lemma 4.1,(1)] $[\![\Theta_{TS_1}]\!]\tilde{\rho}_0 = \mathbf{true}$ $(\tilde{**})$.

The next goal (goal 2) is to show the requirement of consecution for σ_1; as preparation we establish the following lemma.

Lemma A.1 (Edge–transition under variable restriction) *In order to refer to the graph components of a module, let $I_{\mathcal{G}M}$ be the set of indices of the graphs contained in $\mathcal{G}M$; if the context is clear, then instead of $i \in I_{\mathcal{G}M}$ we allow using the shorthand notation $i \in \mathcal{G}M$.*

Under the assumptions of theorem 4.1, let $\tilde{\tau} \equiv_{Def} \tilde{\tau}_e$ be a transition resulting from some edge $e \in \bigcup_{j\in\mathcal{G}M_s} Edges_i$, $s \in \{1,2\}$, $\tau \equiv_{Def} \tau_e$ be the transition resulting from the same edge in $\mathcal{G}M$.

Then for all states $\rho, \rho' \in Val(V_{\mathcal{G}M})$, $\tilde{\rho}, \tilde{\rho}' \in Val(V_{\mathcal{G}M_j})$ with $\tilde{\rho} = \rho_{|V_j}$, $\tilde{\rho}' = \rho'_{|V_j}$ ($V_j \equiv_{Def} V_{\mathcal{G}M_j}$) :

$$\begin{array}{ll} (1) & \tau \text{ is enabled in } \rho \Leftrightarrow \tilde{\tau} \text{ is enabled in } \tilde{\rho} \\ (2) & \rho' \in \tau(\rho) \implies \tilde{\rho}' \in \tilde{\tau}(\tilde{\rho}) \quad . \end{array}$$

Proof . W.l.o.g., we show the correctness for the case $s = 1$ ($\mathcal{G}M_s = \mathcal{G}M_1$). Assume that the edge $e \in Edges_j$, for some $j \in GM_1$, has the form

$$(\ell_1, c \to \bar{x} := \bar{E}, \ell_2)$$

where $\bar{x} = (x_1, \ldots, x_k)$, $\bar{E} = (E_1, \ldots, E_k)$, for some $k \geq 0$; hence

$$\Gamma_e \ : \ \pi_j = \ell_1 \textbf{and} c; x'_1 = E_1, \ldots, x'_k = E_k, \pi'_j = \ell_2$$

Proof of (1) . τ is enabled in ρ
$\Leftrightarrow [\![\pi_j = \ell_1 \textbf{and} c]\!]\rho = \mathbf{true}$
$\Leftrightarrow$ [$free(\pi_j = \ell_1 \textbf{and} c) \subseteq V_1$, lemma 4.1,(1) $\Rightarrow$
$[\![\pi_j = \ell_1 \textbf{and} c]\!]\rho = [\![\pi_j = \ell_1 \textbf{and} c]\!]\tilde{\rho}] \Leftrightarrow [\![\pi_j = \ell_1 \textbf{and} c]\!]\tilde{\rho} = \mathbf{true}$.
$\Leftrightarrow \tilde{\tau}$ is enabled in $\tilde{\rho}$.

Proof of (2) . Assume that $\rho' \in \tau(\rho)$,

$\hookrightarrow$[by def. of $\rho' \in \tau(\rho)$]

(i) $[\![\pi_j = \ell_1 \mathbf{and} c]\!]\rho = \mathbf{true}$;
(ii) $\rho'(x_r) = [\![E_r]\!]\rho, r = 1 \ldots k$;
(iii) $\forall y \in V \backslash \{x_1, \ldots, x_k\} \ : \ \rho'(y) = \rho(y)$.
Show that $\tilde{\rho}' \in \tilde{\tau}(\tilde{\rho})$, by the following sub–goals:
$(\tilde{i})$ $[\![\pi_j = \ell_1 \mathbf{and} c]\!]\tilde{\rho} = \mathbf{true}$.
This follows from (i) as shown in the proof of (1).
$(\tilde{ii})$ $\tilde{\rho}'(x_r) = [\![E_r]\!]\tilde{\rho}, r = 1 \ldots k$.
Assume (ii): $\rho'(x_r) = [\![E_r]\!]\rho, x_r \in V_1, r = 1 \ldots k$;

$\hookrightarrow$[$free(E_r) \subseteq V_1$, $x_r \in V_1$,$\tilde{\rho}' = \rho'_{|V_1}$, $\tilde{\rho} = \rho_{|V_1}$,lemma 4.1,(1) $\Rightarrow$

$[\![E_r]\!]\tilde{\rho} = [\![E_r]\!]\rho = \rho'(x_r) = \tilde{\rho}'(x_r)]$ $\tilde{\rho}'(x_r) = [\![E_r]\!]\tilde{\rho}, r = 1 \ldots k$.
$(\tilde{iii})$ $\forall y \in V_1 \backslash \{x_1, \ldots, x_k\} \ : \ \tilde{\rho}'(y) = \tilde{\rho}(y)$.
Assume (iii), $\forall y \in V \backslash \{x_1, \ldots, x_k\} \ : \ \rho'(y) = \rho(y)$

$\hookrightarrow$[$\tilde{\rho}' = \rho'_{|V_1}$, $\tilde{\rho} = \rho_{|V_1}$, $V_1 \backslash \{x_1, \ldots, x_k\} \subseteq V \backslash \{x_1, \ldots, x_k\} \Rightarrow$

$\forall y \in V_1 \backslash \{x_1, \ldots, x_k\} \ : \ \tilde{\rho}'(y) = \rho'(y) = \rho(y) = \tilde{\rho}(y)]$
$\forall y \in V_1 \backslash \{x_1, \ldots, x_k\} \ : \ \tilde{\rho}'(y) = \tilde{\rho}(y)$.

$\hookrightarrow$[$\tilde{i}$ – $\tilde{ii}\tilde{i}$]

$\tilde{\rho}' \in \tilde{\tau}(\tilde{\rho})$.

q.e.d.

Proof of goal 2. We have to show the requirement of consecution, i.e. that σ_1 is justified by some transition sequence $\sigma\tilde{\tau} =_{Def} (\tilde{\tau}_i)_{i \geq 0}$ of transitions contained in $\mathcal{T}_1 \equiv_{Def} \mathcal{T}_{\mathcal{G}M_1}$, such that $\forall i \geq 0 \ : \ \tilde{\rho}_{i+1} \in \tilde{\tau}_i(\tilde{\rho}_i)$ $(\tilde{***})$.

We derive each transition $\tilde{\tau} \equiv_{Def} \tilde{\tau}_i$ in $\sigma\tilde{\tau}$ from the corresponding transition $\tau \equiv_{Def} \tau_i$ in $\sigma\tau$ $(i \geq 0)$ as follows:

1. If $\tau \equiv \tau_e$ is an edge transition resulting from some graph $\mathcal{G}_j$ in $\mathcal{G}M_1$, defined by the EGA Γ_e over V, then define $\tilde{\tau} =_{Def} \tilde{\tau}_e$ by the same EGA Γ_e, but over the restricted variable set $\mathcal{V}_1$.

2. If $\tau \equiv \tau_I$ (over V), then define $\tilde{\tau} =_{Def} \tau_I$ (over V_1).

3. If $\tau \equiv \tau_E$ (over V), then define $\tilde{\tau} =_{Def} \tau_E$ (over V_1).

4. If $\tau \equiv \tau_e$ is an edge transition resulting from some graph $\mathcal{G}_j$ in $\mathcal{G}M_2$, then define $\tilde{\tau} =_{Def} \tilde{\tau}_E$ (over V_1).

In order to show $\tilde{***}$, let $i \geq 0$ be arbitrarily fixed chosen. Let again $\tilde{\tau} \equiv_{Def} \tilde{\tau}_i$ and $\tau \equiv_{Def} \tau_i$.

Case 1. $\tau \equiv \tau_e$ is an edge transition resulting from some graph $\mathcal{G}_j$ in $\mathcal{G}M_1$.

By assumption, with $\rho' \equiv_{Def} \rho_{i+1}$ and $\rho \equiv_{Def} \rho_i$,

$\rho' \in \tau(\rho)$

$\hookrightarrow$[lemma A.1, $\tilde{\rho} \equiv_{Def} \rho_{|V_1}$, $\tilde{\rho}' \equiv_{Def} \rho'_{|V_1}$]

$\tilde{\rho}' \in \tilde{\tau}(\tilde{\rho})$ $\hookrightarrow$[$\tilde{\rho}' = \tilde{\rho}_{i+1}$ and $\tilde{\rho} = \tilde{\rho}_i$]

$\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$.

Case 2. ($\tau \equiv \tau_I$ over V).

We define $\tilde{\tau} =_{Def} \tau_I$ (over V_1) and have to show that $\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$.

By premise $(***)$, $\rho_{i+1} \in \tau(\rho_i)$.

$\hookrightarrow$[$\tau_I = (\rho \in \mathit{Val}(V) \mapsto \{\rho\})$] $\rho_{i+1} = \rho_i$.

$\hookrightarrow$[$\tilde{\tau} = (\tilde{\rho} \in \mathit{Val}(V_1) \mapsto \{\tilde{\rho}\}), \tilde{\rho}_{i+1} = \rho_{i+1|V_1} = \rho_{i|V_1} = \tilde{\rho}_i$]

$\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$.

Case 3. ($\tau \equiv \tau_E$ over V).

We define $\tilde{\tau} =_{Def} \tau_E$ (over V_1) and have to show that $\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$. We denote the set of variables with mode **external** in $\mathcal{G}M$ by V_E.

By premise $(***)$, $\rho_{i+1} \in \tau(\rho_i)$.

$\hookrightarrow$[$\tau_E = (\rho \in \mathit{Val}(V) \mapsto \{\rho \in V \mid \forall x \in V \backslash V_E : \rho'(x) = \rho(x)\})$]

$\forall x \in V \backslash V_E : \rho_{i+1}(x) = \rho_i(x)$ $(****)$.

Assume that $\tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i)$; denote the set of variables with mode **external** in $\mathcal{G}M_1$ by $V_{1,E}$.

$\hookrightarrow$[$\tilde{\tau}_E = (\tilde{\rho} \in \mathit{Val}(V_1) \mapsto \{\tilde{\rho} \in V_1 \mid \forall x \in V_1 \backslash V_{1,E} : \tilde{\rho}'(x) = \tilde{\rho}(x)\})$]

$\exists x_0 \in V_1 \backslash V_{1,E} \,.\, \tilde{\rho}_{i+1}(x_0) \neq \tilde{\rho}_i(x_0)$.

The relation between $V_{1,E}$ and V_E is as follows: If x is a variable of mode **external** in $\mathcal{G}M$, then it must have had mode **external** in GM_1; hence $V_E \subseteq V_{1,E}$.

$\hookrightarrow$[$V_1 \subseteq V$] $V_1 \backslash V_{1,E} \subseteq V \backslash V_E$.

$\hookrightarrow$[$x_0 \in V \backslash V_E$] $\exists x_0 \in V \backslash V_E \,.\, \rho_{i+1}(x_0) = \tilde{\rho}_{i+1}(x_0) \neq \tilde{\rho}_i(x_0) = \rho_i(x_0)$,

which is a contradiction to $(****)$; hence $\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$ must be true.

Case 4. ($\tau \equiv \tau_e$ is an edge transition resulting from some graph $\mathcal{G}_j$ in $\mathcal{G}M_2$).

We define $\tilde{\tau} =_{Def} \tau_E$ (over V_1) and have to show that $\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$.

Assume that the edge e has the form

$$(\ell_1, c \to \bar{x} := \bar{E}, \ell_2)$$

where $\bar{x} = (x_1, \ldots, x_k)$, $\bar{E} = (E_1, \ldots, E_k)$, for some $k \geq 0$; hence

$$\Gamma_e = :\ \pi_j = \ell_1 \mathbf{and} c; x'_1 = E_1, \ldots, x'_k = E_k, \pi'_j = \ell_2$$

Since $\rho_{i+1} \in \tau(\rho_i)$,
(i) $[\![\pi_j = \ell_1 \mathbf{and} c]\!]\rho_i = \mathbf{true}$;
(ii) $\rho_{i+1}(x_r) = [\![E_r]\!]\rho_i, x_r \in V_2(!), r = 1 \ldots k$;
(iii) $\forall y \in V \backslash \{x_1, \ldots, x_k\}\ :\ \rho_{i+1}(y) = \rho_i(y)$.

Assume that $\tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i)$ (again, denote the set of variables with mode **external** in $\mathcal{G}M_1$ by $V_{1,E}$).

$$\hookrightarrow [\tilde{\tau}_E = (\tilde{\rho} \in \mathit{Val}(V_1) \mapsto \{\tilde{\rho} \in V_1\ \mid \forall x \in V_1 \backslash V_{1,E}\ :\ \tilde{\rho}'(x) = \tilde{\rho}(x)\})]$$

$$\exists x_0 \in V_1 \backslash V_{1,E}\ .\ \tilde{\rho}_{i+1}(x_0) \neq \tilde{\rho}_i(x_0)\ (*****).$$

The modes of the variables $\{x_1, \ldots, x_k\}$ must be either **out** or **local**; one of these variables could be declared in $\mathcal{G}M_1$ only with the mode **external**

$$\hookrightarrow [x_0 \in V_1 \backslash V_{1,E}]\ \ x_0 \notin \{x_1, \ldots, x_k\}.$$

$$\hookrightarrow [\text{(iii)}]\ \ \rho_{i+1}(x_0) = \tilde{\rho}_{i+1}(x_0) = \tilde{\rho}_i(x_0) = \rho_i(x_0),$$

which is a contradiction to $(*****)$; hence $\tilde{\rho}_{i+1} \in \tilde{\tau}(\tilde{\rho}_i)$ must be true.

Fairness. In order to establish that $\sigma_1 \in L(\mathcal{G}M_1)$, $\sigma_1 \equiv_{Def} (\tilde{\rho}_i)_{i \geq 0}$, it remains to show that σ_1 is $\mathcal{G}M_1$–fair, i.e. fair with respect to the sets $\mathcal{J}_1$ and $\mathcal{C}_1$ of the transition system $TS_1 \equiv_{Def} TS_{\mathcal{G}M_1}$ which defines the semantics of $\mathcal{G}M_1$.

We will show the opposite direction: From the assumption that σ_1 does *not* conform to either of the fairness requirements in TS_1, it follows that σ_0, $\sigma_0 \equiv_{Def} (\rho_i)_{i \geq 0}$, also violates an according fairness requirement in $TS \equiv_{Def} TS_{\mathcal{G}M}$ (where $\sigma_1 = \sigma_{0|V_1}$).

A violation of one of the fairness requirements in TS_1 can happen only, if there is a transition $\tilde{\tau} \in \mathcal{T}_1$ ($\mathcal{T}_1 \equiv_{Def} \mathcal{T}_{TS_1}$), which is enabled during the computation σ_1 at infinitely many moments $i \in I$, for some infinite set of moments $I \subseteq N_0$ ($\mathit{Time} = N_0$). Furthermore, $\tilde{\tau} \equiv_{Def} \tilde{\tau}_e$ must be an edge transition resulting from some edge $e \in \mathit{Edges}_j$, $j \in \mathcal{G}M_1$.

Consider the transition $\tau \equiv_{Def} \tau_e$ derived from the same edge in $\mathcal{G}M$ (over V). Then by lemma A.1,(1)

$$\forall i \geq 0\ :\ \tilde{\tau}(\tilde{\rho}_i) \neq \emptyset \Leftrightarrow \tau(\rho_i) \neq \emptyset \qquad (ENB_\tau)$$

i.e. $\tilde{\tau}$ is enabled at the same moment during σ_1 where τ is enabled during σ_0.

Justice. Assume that $\tilde{\tau} \in \mathcal{J}_1 \subseteq \mathcal{T}_1$ violates the requirement of justice over σ_1, i.e. from some moment $i_0 \geq 0$ on, $\tilde{\tau}$ is permanently enabled over σ_1, but not taken:

$$\forall i \geq i_0\ :\ \tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i) \wedge \tilde{\tau}(\tilde{\rho}_i) \neq \emptyset \quad .$$

By definition of TS_1,

$$J_1 = \{\tilde{\tau}_e \mid e \in \bigcup_{j \in \mathcal{G}M_1} Edges_j \backslash E\bar{\mathcal{J}}_j\}$$

i.e. $\tilde{\tau} \equiv_{Def} \tilde{\tau}_e$ for some edge e in $\mathcal{G}M_1$.
Consider the transition $\tau \equiv_{Def} \tau_e$ derived from the same edge in $\mathcal{G}M$ (over V).

$\hookrightarrow$[$\mathcal{J} = \{\tau_e \mid e \in \bigcup_{j_1 \in \mathcal{G}M_1} Edges_{j_1} \backslash E\bar{\mathcal{J}}_{j_1} \cup \bigcup_{j_2 \in \mathcal{G}M_2} Edges_{j_2} \backslash E\bar{\mathcal{J}}_{j_2}\}$]

$\tau \in \mathcal{J}$.

We show that τ would violate the requirement of justice over σ_0.

$\forall i \geq i_0 \ : \ \tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i) \wedge \tilde{\tau}(\tilde{\rho}_i) \neq \emptyset$

$\hookrightarrow$[by lemma A.1,(2): $\tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i) \Rightarrow \rho_{i+1} \notin \tau(\rho_i)$, and ENB_τ]

$\forall i \geq i_0 \ : \ \rho_{i+1} \notin \tau(\rho_i) \wedge \tau(\rho_i) \neq \emptyset$

hence $\sigma_0 \notin L(\mathcal{G}M)$, by contradiction to the premise.

Compassion. Assume that $\tilde{\tau} \in \mathcal{C}_1 \subseteq \mathcal{T}_1$ violates the requirement of compassion over σ_1, i.e. $\tilde{\tau}$ is enabled during the computation σ_1 at infinitely many moments $i \in I$, for some infinite set of moments $I \subseteq N_0$ ($Time = N_0$), but from some moment $i_0 \geq 0$ on, $\tilde{\tau}$ is never taken:

$$\forall i \geq i_0 \ : \ \tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i) \text{ , and } \forall i \in I \ : \ \tilde{\tau}(\tilde{\rho}_i) \neq \emptyset \quad .$$

By definition of TS_1,

$$C_1 = \{\tilde{\tau}_e \mid e \in \bigcup_{j \in \mathcal{G}M_1} E\mathcal{C}_j\}$$

i.e. $\tilde{\tau} \equiv_{Def} \tilde{\tau}_e$ for some edge e in $\mathcal{G}M_1$.
Consider the transition $\tau \equiv_{Def} \tau_e$ derived from the same edge in $\mathcal{G}M$ (over V).

$\hookrightarrow$[$\mathcal{C} = \{\tau_e \mid e \in \bigcup_{j_1 \in \mathcal{G}M_1} E\mathcal{C}_{j_1} \cup \bigcup_{j_2 \in \mathcal{G}M_2} E\mathcal{C}_{j_2}\}$]

$\tau \in \mathcal{C}$.

We show that τ would violate the requirement of compassion over σ_0.

$\forall i \geq i_0 \ : \ \tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i) \wedge \forall i \in I \ : \ \tilde{\tau}(\tilde{\rho}_i) \neq \emptyset$

$\hookrightarrow$[by lemma A.1,(2): $\tilde{\rho}_{i+1} \notin \tilde{\tau}(\tilde{\rho}_i) \Rightarrow \rho_{i+1} \notin \tau(\rho_i)$, and ENB_τ]

$\forall i \geq i_0 \ : \ \rho_{i+1} \notin \tau(\rho_i)$, and $\forall i \in I \ : \ \tau(\rho_i) \neq \emptyset$

hence $\sigma_0 \notin L(\mathcal{G}M)$, by contradiction to the premise.

q.e.d.

Appendix B

Remarks on chapter 3

In the following, we present supplementary material which does not belong to the mainstream of the thesis.

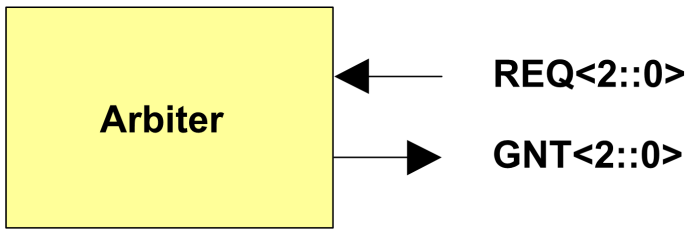
B.1 Note on the assertion language

We have motivated in chapter 2 the introduction of the visual formalism STD as a method for requirement specification and (finite) model verification, using a verification environment based on model–checking.

High–level modelling languages, for example VHDL or Statemate^{TM}, use rich sets of basic and composite data–types. For example, a bundle of hardware wires is typically modelled in VHDL as an array of signals of type Bit. Consider a simple arbitration unit in isolation (this unit is taken from the bus–bridge example used earlier in chapter 2, see figure B.2).

The interface of this unit is depicted in figure B.1.

Figure B.1: Interface of arbiter unit.



The state of each of the interface signals GNT[0] ... GNT[2] at a given moment in (simulation) time is given by a corresponding assertion GNT[i] = x ($i = 0 \ldots 2$) , where x is the constant '0' or '1' (assuming a two–valued domain for type Bit). A sample informal statement of a property of (the interface signals of) this device is:

$$\textbf{always}\ (\langle\!|GNT[0] = \text{'1'}|\!\rangle \rightarrow \textbf{eventually}\ \langle\!|GNT[0] = \text{'0'}|\!\rangle)$$

The operator $\langle\!| \ldots |\!\rangle$ is a type–cast of the domain of Boolean values (the result type of the comparison operator =) into the domain of truth values.

The concrete syntax of expressions (in particular Boolean expressions, which we use to specify sets of system states) has no relevance for our framework.

What *is* of importance, however, is the visibility of signals; only signals which are part of a component interface may be included in specification expressions.

B.2 Note on first–order specifications

Our definition of temporal logic (LTL_V) does not allow any form of quantification. In practice, the lack of expressive power due to missing quantification in the specification logic can be addressed by another method, namely the introduction of so–called *rigid specification variables*.

A rigid specification variable can be considered as a further unconnected input, which has a random value at the beginning of each system run. Once the system starts running, the value of the new signal remains fixed.

Using this technique, we can write the property stated above in a more general way, introducing a rigid specification variable linenr (cf. figure B.3), in the following way (using ad–hoc syntax declare ... require ...):

$$\textsf{declare}\ \text{rigid}\ linenr : \text{integer}\ 0\ ..\ 2;$$
$$\textsf{require}\ \textbf{always}\ (\langle\!|GNT[linenr] = \text{'1'}|\!\rangle \rightarrow \textbf{eventually}\ \langle\!|GNT[linenr] = \text{'0'}|\!\rangle)$$

It is easy to see that the effect of model–checking of this specification is the verification of the following property with quantification:

$$\forall linenr : \textbf{always}\ (\langle\!|GNT[linenr] = \text{'1'}|\!\rangle \rightarrow \textbf{eventually}\ \langle\!|GNT[linenr] = \text{'0'}|\!\rangle)$$

The use of rigid specification variables has been included in the design of

Figure B.2: Structure of a PCI–to–host bus–bridge device.

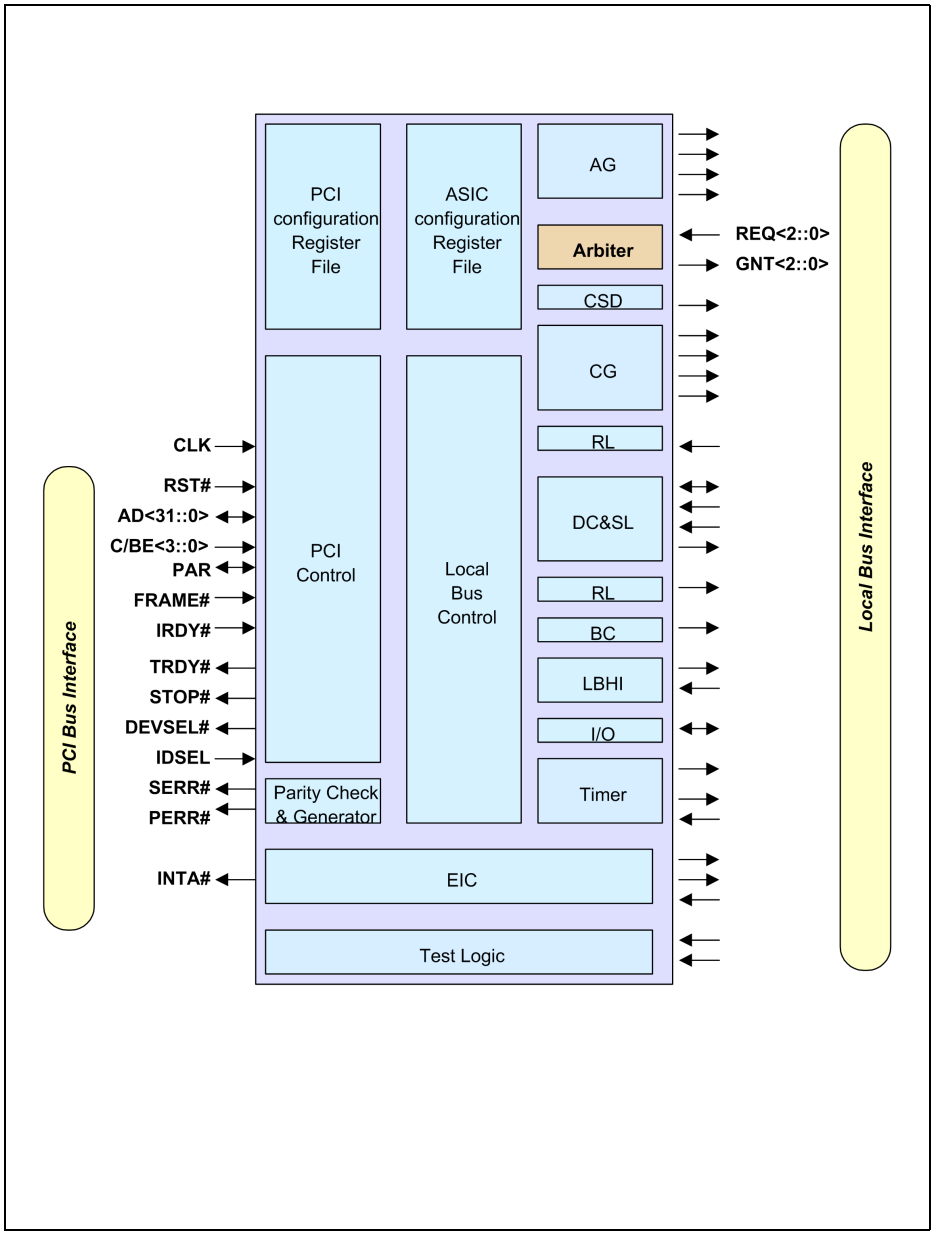
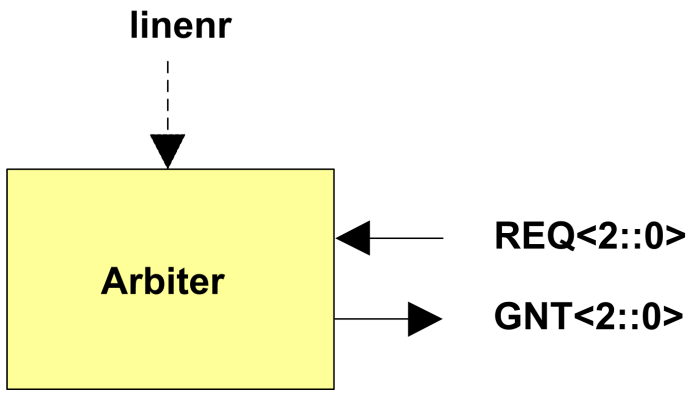


Figure B.3: Extended interface arbiter unit, with auxiliary input–signal linenr.



STDx, making the model–checking technique more powerful. Note however that compositional reasoning using specification variables is usually very complicated; moreover, verification results can often be obtained much easier using reasoning about the model semantics (in particular, using abstraction).

B.3 Notes on $LINLTL_V$

We have introduced different temporal logics and notions in chapter 3. The technical devices of chapter 3 are needed in the first place to explain the semantics of LSTD and STD (chapter 5 and 6). Nevertheless, the logic expert might be interested to learn more about the logics themselves.

This appendix is not a complete account on the nature of the logic formalisms introduced in chapter 3; such an investigation was outside the scope of this thesis. Our aim is here to outline the main properties of the sub–logics of *LTL* and to add some illustrative example.

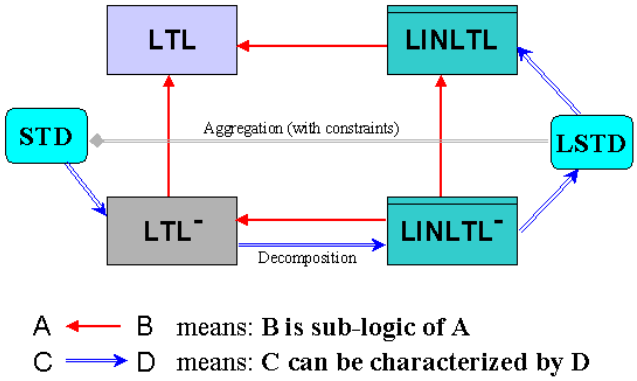
The figure B.4 recalls, that LTL_V^- and $LINLTL_V$ are subsets (sub–logics) of the standard (linear–time) temporal logic LTL_V.

The logic $LINLTL_V^-$ is in turn a sub–logic of LTL_V^-. As corollary 3.1 has shown, for each formula in LTL_V^- there is an equivalent formula in $LINLTL_V^-$; hence the expressive power of these logics is the same.

In this appendix we will further discuss two aspects:

- How the logic $LINLTL_V$ evolves naturally in the context of the semantics of LSTD

Figure B.4: Overview of the relationship of the logic formalisms introduced in this thesis.



- how first–order aspects are handled in the logics, as well as in STD.

The structure of the logic $LINLTL_V$ (introduced in chapter 3, def. 3.14) reflects two different semantic paradigms:

- declarative – through the initial/invariant semantics
- operational – through the nesting structure of **unless** (**until**) combinations, where nesting occurs only in the second argument of **unless** (**until**).

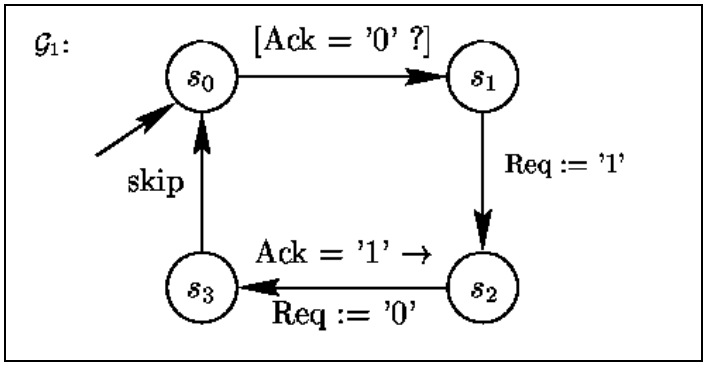
We rephrase an example given in chapter 6 in temporal logic notation.

Example B.1 (Phase–level specification of master–protocol) *Recall the definition of the master–module introduced in chapter 5 :*

$$\begin{aligned} \mathcal{G}M_{Req} : \quad & \textbf{module} \ \text{Master} \\ & \textbf{out} \ \text{Req} : \text{Bit} \ \textbf{where} \ \text{Req} = \text{'0'} \\ & \textbf{external} \ \text{Ack}: \text{Bit} \\ & \mathcal{G}_1 \end{aligned}$$

where $\mathcal{G}_1$ is defined as shown in figure B.5.

Figure B.5: Implementation of Master–model of the basic 4–phase handshake protocol.



Then the implementation shown in figure B.5 satisfies the following local specification with respect to the interface variables Req, Ack :

$$\phi^M \equiv_{Def} \phi_0^M \wedge \phi_{11}^M \wedge \phi_{12}^M \wedge \phi_{13}^M \wedge \phi_{14}^M$$

where

Initialization:

$$\phi_0^M \equiv_{Def} \quad \phi_{0,1} \rightarrow \phi_{0,0}$$

Invariance:

$$\begin{aligned}
\phi_{11}^M &\equiv_{Def} \quad \mathbf{always}\,(\phi_{0,0} \rightarrow \mathbf{next}\,(\phi_{0,0}\,\mathbf{unless}\,(\phi_{1,0} \vee \phi_{X,1}))) \\
\phi_{12}^M &\equiv_{Def} \quad \mathbf{always}\,(\phi_{1,0} \rightarrow \mathbf{next}\,(\phi_{1,0}\,\mathbf{unless}\,\phi_{1,1})) \\
\phi_{13}^M &\equiv_{Def} \quad \mathbf{always}\,(\phi_{1,1} \rightarrow \mathbf{next}\,(\phi_{1,1}\,\mathbf{until}\,(\phi_{0,1} \vee \phi_{X,0}))) \\
\phi_{14}^M &\equiv_{Def} \quad \mathbf{always}\,(\phi_{0,1} \rightarrow \mathbf{next}\,(\phi_{0,1}\,\mathbf{unless}\,\phi_{0,0}))
\end{aligned}$$

and

$$\phi_{x,y} \equiv_{Def} \langle\!|Req = x|\!\rangle \wedge \langle\!|Ack = y|\!\rangle \quad .$$

The specification style of example B.1 is common for temporal logic specifications. A similar style is also used in UNITY specifications [9]. On the other hand, the graph structure of figure B.5 suggests a specification, where the it-

eration points of the protocol (4–phase Req/Ack handshake) are highlighted. For our example, the iteration point is characterized by the state assertion $\phi_{0,0}$.

This leads to a specification style, where different specification fragments are combined in one specification clause, which is essentially the idea of waveforms in LSTD and STD.

This leads to the structure of $LINLTL_V$.

Example B.2 (Phase–level specification of master–protocol, using $LINLTL_V$) *With respect to example B.1 , the following specification is equivalent:*

$$\phi^M_{lin} \equiv_{Def} \phi^M_0 \wedge \phi^M_1$$

where

Initialization:

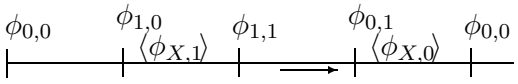
$$\phi^M_0 \equiv_{Def} \quad \phi_{0,1} \rightarrow \phi_{0,0}$$

Invariance:

$$\begin{aligned}\phi^M_1 \equiv_{Def} \quad & \mathbf{always}\,(\phi_{0,0} \rightarrow \\ & \mathbf{next}\,(\phi_{0,0}\ \mathbf{unless}\,(\phi_{1,0} \wedge \\ & \mathbf{next}\,(\phi_{1,0}\ \mathbf{unless}\,(\phi_{1,1} \wedge \\ & \mathbf{next}\,(\phi_{1,1}\ \mathbf{until}\,(\phi_{0,1} \wedge \\ & \mathbf{next}\,(\phi_{0,1}\ \mathbf{unless}\,\phi_{0,0})) \vee \phi_{X,0}))) \\ & \vee \phi_{X,1})))\end{aligned}$$

Note that the formula ϕ^M_1 is equivalent to the LSTD diagram $\alpha\Delta^M_1$ introduced in chapter 5.

$\alpha\Delta^M_1 \equiv_{Def}$



(cf. lemma 5.2.)

Appendix C

Notations

$\rho \in Val(V)$	valuation	Def. 3.2
$\mathcal{A}$	symbolic automaton	Def. 3.3
$\sigma \in Comp(V)$	computation	
$\sigma\ell$	run	Def. 3.5
$L(\mathcal{A})$	sem. of A	Def. 3.5
any x . $x \in M$	**any** –operator	Lem. 3.2
$\phi \in LTL_{A\mathcal{L}}$	lin. temp. logic formula	Def. 3.8
$\sigma \models \phi$	satisfies	Def. 3.9
Φ	formula scheme	Def. 3.10
ξ	sequencing LTL formula	Def. 3.16
$\tau \in \mathcal{T}$	transition	Def. 4.1
$\mathcal{G}$	transition graph	Def. 4.3
$\mathcal{G}S$	trans. graph system	Def. 4.4
$TS_{\mathcal{G}S}$	transition system	Def. 4.5

$\mathcal{G}M$	open trans. graph system	Def. 4.6
$\mathcal{G}M_1 \| \mathcal{G}M_2$	module composition	Def. 4.9
$\mathcal{K}$	Kripke structure with fairness	Def. 4.10
Δ	LSTD–body	Def. 5.1
ΔE	LSTD–phase	Def. 5.1
$\alpha\Delta$	LSTD–diagram	Def. 5.2
αE	activation spec.	Def. 5.2
ΔS	LSTD–specification	Def. 5.5
$\mathcal{W}$	bundle of waveforms	Def. 6.1
$W\Delta$	STD–diagram	Def. 6.1
ζ	timeline–position	Def. 6.5

Bibliography

[1] A. Allara, M. Bombana, S. Comai, B. Josko, R. Schlör, and D. Sciuto. Specification of embedded monitors for property checking. In *Proceedings, Forum on Design Languages, FDL'99*, pages 117–126, 1999.

[2] ANSI/IEEE Std 1076–1987. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, USA, March 1988.

[3] C. Antoine and B. Le Goff. Timing diagrams for writing and checking logical and behavioral properties of integrated systems. In P.Prinetto and P.Camurati, editors, *Correct Hardware Design Methodologies*, pages 441–453. Elsevier Science Publishers B.V., 1992.

[4] B. Josko. *Modular Specification and Verification of Reactive Systems*. Habilitationsschrift, Universität Oldenburg, 1993.

[5] T. Bienmüller, J. Bohn, H. Brinkmann, U. Brockmeyer, W. Damm, H. Hungar, and P. Jansen. Verification of automotive control units. In Ernst-Rüdiger Olderog and Bernd Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 319–341. Springer Verlag, 1999.

[6] U. Brockmeyer. *Verifikation von STATEMATE Designs*. Dissertation, Fachbereich Informatik, Universität Oldenburg. Berichte aus dem Fachbereich Informatik; Nr.16, 1999.

[7] J. R. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel et.al., editor, *Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the fifth annual Symposium on Logics in Computer Science*, pages 428–439, June 1990.

[9] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[10] CVE (Circuit Verification Environment): User's Guide. Siemens AG, 1997.

[11] W. Damm, G. Döhmen, V. Gerstner, and B. Josko. Modular verification of petri nets: The temporal logic approach. In Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness*, Lecture Notes in Computer Science, 430, pages 180–207. Springer-Verlag, 1990.

[12] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 293–312, 1999.

[13] W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based system-level hardware designs. In E. Börger, editor, *Specification and Validation Methods*, pages 331–409. Oxford University Press, 1995.

[14] K. Feyerabend and B. Josko. A visual formalism for real time requirement specifications. In Miquel Bertran and Teodor Rus, editors, *Transformation-Based Reactive Systems Development, Proceedings, 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97*, volume 1231 of *Lecture Notes in Computer Science*, pages 156–168. Springer-Verlag, 1997.

[15] N. Francez. *Fairness*. Springer-Verlag, 1987.

[16] M. Fujita and H. Fujisawa. Specification, verification and synthesis of control circuits with propositional temporal logic. In J.A.Darringer and F.J.Rammig, editors, *Computer Hardware Description Languages and their Applications*, pages 265–279. Elsevier Science Publishers B.V., 1990.

[17] D. Harel, A. Pnueli, J.P. Schmidt, and R.Sherman. On the Formal Semantics of Statecharts. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[18] J. Helbig. *Linking Visual Formalisms: A Compositional Proof System for Statecharts Based on Symbolic Timing Diagrams*. Dissertation, Fach-

bereich Informatik, Universität Oldenburg. Berichte aus dem Fachbereich Informatik; Nr.4, 1998.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Journal ACM*, 12:576–580, 1969.

[20] i-Logix, 22 Third Avenue, Burlington, Mass. 01803, USA. *Languages of STATEMATE*, January 1991.

[21] K. Fisler. A logical formalization of hardware design diagrams. Technical Report 416, Indiana University Computer Science Department, September 1995.

[22] Ch. Kleuker. *Constraint Diagrams*. Dissertation, Fachbereich Informatik, Universität Oldenburg. Berichte aus dem Fachbereich Informatik; Nr.3, 2000.

[23] F. Korf. *System-Level Synthesewerkzeuge: Von der Theorie zur Anwendung*. Dissertation, Fachbereich Informatik, Universität Oldenburg, 1997.

[24] Fred Kröger. *Temporal Logic of Programs*. EATCS-Monographs. Springer-Verlag, 1987.

[25] K. Lüth. The ICOS Synthesis Environment. In A.P. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *LNCS*, pages 294–297. Springer Verlag, 1998.

[26] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems (Specification)*. Springer-Verlag, 1992.

[27] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[28] PCI local bus specification, rev. 2.0. PCI Special Interest Group, 1993.

[29] S. Rakitin. *Software verification and validation: A practitioners guide*. ARTECH HOUSE, INC., 1997.

[30] R. Schlör. A Prover for VHDL–based Hardware Design. In *IFIP International Conference on Computer Hardware Description Languages and their Applications*, pages 643–650, Sep. 1995.

[31] R. Schlör, A. Allara, and S. Comai. System Verification using User-Friendly Interfaces. In *Design, Automation and Test in Europe / User Forum*, pages 167–172. IEEE Computer Society Press, 1999.

[32] R. Schlör and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proceedings, The European Conference on Design Automation*, pages 518–524. IEEE Computer Society Press, 1993.

[33] R. Schlör, B. Josko, and D. Werth. Using a visual formalism for design verification in industrial environments. In Tiziana Margaria, editor, *services and visualization*, volume 1385 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, 1998.

[34] VIS: Verification Interacting with Synthesis. `http://www-cad.eecs.berkeley.edu/Respep/Research/vis`, 1995.

[35] W.-D. Tiedemann. Synthese von synchronen Steuerwerken mit Echtzeitbedingungen. In W. Grass and M. Mutz, editors, *3. GI/ITG Workshop zur Anwendung formaler Methoden beim Entwurf von Hardwaresystemen*, Berichte aus der Informatik, pages 21–31. Shaker Verlag, 1995.

[36] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.

[37] Thomas Wilke. Classifying discrete temporal properties. In Christoph Meinel, editor, *STACS'99*, volume 1563 of *Lecture Notes in Computer Science*, pages 32–46. Springer Verlag, 1999.

Rainer C. Schlör

married with Elsa Funk–Schlör

two children (Sebastian and Tabea Schlör)

Curriculum Vitae

21.09.1960	born in Wuppertal, Germany
1970–1979	school (Wilhelm–Dörpfeld–Gymnasium Wuppertal)
1981–1987	study of computer science at the Rheinisch Westfälische Universität Aachen. The study was partially supported by a scholarship of the Studienstiftung des Deutschen Volkes
1989–2000	member of the research group of Prof. Dr. Werner Damm at the Carl–von–Ossietzky Universität Oldenburg and the research institute OFFIS (since 1993)
March 2, 2001	defense of the thesis
April 25, 2001	joined McKinsey&Company, Munich, as practice specialist

*** last page ***